SPRINGER REFERENCE

Soonhoi Ha
Jürgen Teich
Editors

Handbook of Hardware/ Software Codesign

Springer

Handbook of Hardware/Software Codesign

Soonhoi Ha • Jürgen Teich
Editors

Handbook of Hardware/Software Codesign

With 575 Figures and 56 Tables

Springer

Editors
Soonhoi Ha
Department of Computer
Science and Engineering
Seoul National University
Seoul, Korea

Jürgen Teich
Department of Computer Science
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
Erlangen, Germany

ISBN 978-94-017-7266-2 ISBN 978-94-017-7267-9 (eBook)
ISBN 978-94-017-7268-6 (print and electronic bundle)
https://doi.org/10.1007/978-94-017-7267-9

Library of Congress Control Number: 2017947685

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer Science+Business Media B.V.
The registered company address is: Van Godewijckstraat 30, 3311 GX Dordrecht, The Netherlands

Foreword

Hardware/software codesign means to achieve system-level design objectives by leveraging the synergy between hardware and software through their concurrent design. Codesign has been practiced in various ways since the inception of digital systems. The specification of *instruction-set architectures* enabled the concurrent development of hardware and software as well as the creation of high-level languages and compilers. Grace Hopper was indeed a pioneer of codesign in the early 1950s with the design of portable languages (i.e., machine-independent), which led to the development of COBOL and of modern programming languages.

Within about 70 years of computer science and engineering, various embodiments of abstractions, programmability, and hardware have given different meanings to hardware/software codesign. The renewed interest on this topic in the last two decades relates to the use of structured design methodologies and tools for hardware and software design. Thus, *electronic systems design automation* had to embrace hardware/software codesign as one of its major tasks and objectives. The formalization of the related design problems enabled synthesis and verification of hardware/software systems through the development of computer-aided design methods and tools.

However, it is our opinion that a sound system design methodology must start by capturing the design specifications at the highest level of abstraction and then proceed toward an efficient implementation by subsequent refinement steps. The partition of the design into hardware and software is indeed a consequence of decisions taken at a higher level of abstraction. The critical decisions are about the architecture of the system (processors, buses, hardware accelerators, memories, and so on) that will carry on the computation and communication tasks associated with the overall specification of the design. This design process is segmented into a series of similar steps. The principles at the basis of each step consist of hiding unnecessary details of an implementation, summarizing the important parameters of the implementation in an abstract model, and limiting the design space exploration to a set of potential platform instances. The design process is a meet-in-the-middle approach where the refinement from specification toward implementation is matched against a library of components whose models are abstractions of possible implementations.

This was indeed the basis for the development of a design methodology that goes under the name of platform-based design where the steps outlined above have been formalized wherever possible. This methodology includes the development of hardware and the related software when the architectural decisions have been made and the design tasks have been mapped to the components of the architecture. A task that is mapped into hardware can then be synthesized with the appropriate tools in parallel to the software development that takes place when the mapping process has allocated the task to a programmable component. Note also that in this framework, it is rather clear that according to the available programmable components, different software design processes can be developed. In fact, programming a microprocessor is quite different than programming a DSP or a special purpose processor.

The first step in the design process is then capturing a set of specifications or requirements on the functionality and the architecture of the design. These will guide the design process through the refinement steps. Requirements are in general denotational statements about what the system is supposed to do. For example, if we are to design a special purpose math processor that computes the solution of nonlinear algebraic equations, the functional requirement would be stated as:

Find x such that $F(x) = 0$,

where no algorithm to accomplish this task has been chosen. The choice of the algorithm is already a refinement step in the design task. This example underlines that requirements are abstract statements about what the design has to accomplish. Some of the requirements may be given in terms of the properties of an implementation but still in abstracted form. For example,

The system has to consume no more than 1kW of power.

Of course this is a constraint that encompasses the entire design space from functionality to final implementation, and while in the first steps of the architecture selection, the power consumption can be estimated, and it will have to be verified at the final implementation where the physics of the solution will be known. The design space exploration is determined in part by these requirements.

In addition to the requirements, often a set of desirable features of the design can be stated. In this case, the mathematical formalism is a function that can be either minimized or maximized. Then the refinement steps take the form of an optimization problem where the objective functions are optimized in the presence of complex constraints.

Often the design process of interest has already been given in terms of high-level functionality where some design decisions have been taken. Using the example above, we may be asked to implement the Newton–Raphson procedure, a choice for the algorithm to be used to meet the requirement. This is given in terms of behavior, i.e., an operational description at the appropriate level of abstraction.

Once the behavior has been selected and described, then it is time to determine an architecture to implement this behavior that optimizes the goal function(s) and satisfies the constraints. The architecture may be developed anew or obtained combining elements in a library of available components or a combination of both

whereby library elements are combined with virtual components that have to be designed from scratch.

In the design process, synthesis steps are intertwined with verification steps that check whether the constraints are satisfied, the functionality is correctly implemented, and the design is feasible.

This handbook covers extensively many topics specific to hardware/software codesign intended as system design as described above, namely, modeling, design and optimization, validation and verification, as well as application areas. Modeling has been a key design technology for capturing system-level aspects: it is achieved today via specialized languages and graphical formalisms. The underlying semantics of these representations is key to the application of rigorous methods to capture the real intent of the design and to offer a framework where properties of the design can be assessed. At the same time, the expressive power of the language is important to serve a wide variety of designers and design applications. For example, within general-purpose languages, SystemC – a class library with hardware semantics – has shown to be a viable extension to C++ to capture hardware components in an object-oriented fashion.

System architectures have changed significantly over the last two decades, to exploit the growth and diversification of the underlying semiconductor technology. As a result of the limited growth of clock operational frequencies and the wide availability of devices due to downscaling, multiprocessor architectures with significant on-chip memory (or low-latency off-chip memory) are dominating the market. Indeed, multiprocessing fits the need of realizing systems with limited energy consumption, thus avoiding thermal and dark silicon issues. Codesign in a multiprocessing environment provides major challenges, such as exploration of the design space and of parametric choices that can maximize the return of distributed software applications. Design and optimization require often cross-layer techniques that can span various modeling abstractions and operate on the tuning of various system aspects concurrently.

Much research emphasis on memory architectures has been fueled both by the need to handle big data "in proximity" as well as by the availability of novel memory technologies including their physical stacking. It is important to remark that this problem is not only a hardware design problem, as the potential beneficial use of memory hierarchies affects system and software design. By the same token, on-chip communication has evolved to *networks-on-chips (NoCs),* which encompass various structured interconnect schemes leveraging data packetization and routing. NoC design within multiprocessing systems requires the use of specific design techniques to match hardware structures realizing the network architecture to their operational protocols that are often programmable and specified in software.

Many design tools have been proposed to synthesize, partition, and optimize systems. In the recent years, the use of programmable processor cores (e.g., ARM) as black boxes within multiprocessing systems has led to a specific focus on both memory and communication synthesis and optimization. Conversely, the search for energy-performance optimal computational engines has led to *application-specific instruction-set processors*. Such processors occupy a limited but strategic part of

the computing product spectrum and pose a key codesign problem. Indeed, the definition of an instruction set has been the fulcrum of codesign techniques since the invention of the digital computer. Thus, the possibility of designing and optimizing the instruction set can be viewed as searching for an optimum position of this fulcrum to balance the hardware and software cost and performance.

The selection of the functionalities to implement in hardware and of the ones in software is a system design issue that precedes HW/SW codesign. Indeed, system design can be characterized as function/architecture codesign, where function is what we wish to realize and architecture is how we are going to implement the functionality. As described above, architecture can be defined as the functional level as well. In this case, we decompose a function into a network of subfunctions. Each of this subfunction can be further decomposed until we decide to allocate the leaves of the functional decomposition to components of a hardware architecture. The hardware architecture consists of components such as processors, memories, sensors, actuators, communication entities, and specialized hardware components. Once a block of functionality is assigned to a programmable component, its implementation will be a software program running onto that component. If it is assigned to a specialized hardware, then its implementation will be a set of IP blocks, and we have a HW/SW codesign problem at hand. System design is where important decisions are taken and where it is of paramount importance to consider available components to maximize reuse. *Platform-based design* has been a major step forward in conceiving HW/SW systems that enabled the use of synthesis and verification tools with high efficiency. Indeed, a platform is a restriction of the design space.

Methods and tools for software synthesis and optimization have led to the automatic rewriting of specification in terms of the best primitives to be used by a processor. For example, ARM processors benefit from using guarded instructions, and making them explicit in software improves the compiler performance. Software analysis – in terms of execution time – is extremely important to quantify and bound delay in system design, especially in view of satisfying timing constraints for task executions. Thus, software timing analysis and verification is a key task of HW/SW codesign.

Validating system design is the most important task of all, since most digital systems are required to satisfy safety and dependability constraints. An important area is the verification of formal models that abstract parts – if not the entirety – of digital systems. Formal verification is based on choosing specific properties and checking if they are satisfied in all operational instances. Functional and timing behavior are cornerstones of verification. Often such properties are shown to hold with subsystems, and thus system composability is a key asset in proving correctness by construction. Needless to say, few systems are composable in a straightforward way, and this motivates the large research effort in verification. Large systems are often validated by semiformal techniques or by broader but weaker techniques such as simulation, emulation, and prototyping. The inherent weakness of these techniques is in asserting properties that are valid under a wide set of environmental conditions. Unfortunately, when systems fail, they often fail under unusual operating conditions.

Codesign is practiced differently in various application domains. This book covers examples such as datacenter, automotive system, video/image processing, and cyber-physical system design. The peculiarities of these domains in terms of requirements and objectives are reflected in the various ways of applying codesign modeling abstraction as well as synthesis, optimization, and verification methods.

Overall, this handbook presents a broad set of techniques that show the inherent maturity of the state of the art in hardware/software codesign.

University of California at Berkeley
USA
November 2016

Alberto L. Sangiovanni-Vincentelli

Institute of Electrical Engineering
EPFL, Switzerland
November 2016

Giovanni De Micheli

Preface

Hardware/software (HW/SW) codesign was first introduced as a new design methodology for SoCs (systems-on-chip) in the early 1990s to design hardware and software concurrently with the goal to reduce the design time and cost of such systems. After more than 25 years of incessant research and development, it is now regarded as a de facto standard, and the term has become serving as an umbrella for methodologies to design complex electronic systems, even distributed embedded systems. HW/SW codesign covers the full spectrum of system design issues from initial behavior specification to final implementation. Codesign methodologies also include modeling the system behavior independently of the system architecture at a high level and exploring the design space of system architecture at the early design stage. For fast design space exploration, it is necessary to estimate the system performance and resource requirements. HW/SW cosimulation enables us to develop software before hardware implementations become available. Finally, cosynthesis denoting the process of automatically synthesizing hardware components as well as software from a given specification for implementation on a target platform and including also the interfaces for communication between hardware components and processors belongs to the key problems attacked by codesign.

In spite of its significance and usefulness, we discovered that it is quite difficult to understand and learn about its benefits and full impact on real system design, particularly because there did not exist any book or reference on HW/SW codesign until the time of writing this book. Thus, it is our great pleasure to edit this handbook, quenching the thirst for the reference. In this book, we present to you the core issues of hardware/software codesign and key techniques in the design flow. In addition, selected codesign tools and design environments are described as well as case studies that demonstrate the usefulness of HW/SW codesign. This book will be updated regularly to follow the progress of design techniques and introduce commercial as well as research design tools available for our readers. It is meant to serve as a reference not only to interested researchers and engineers in the field but

equally to students. We hope you all will grasp the wide spectrum of subjects that belong to HW/SW codesign and get most benefits out of it for your system design and related optimization problems.

Department of Computer Science and Engineering
Seoul National University
Gwanak-ro 1, Gwanak-gu
Seoul, Korea
June 2017

Soonhoi Ha

Department of Computer Science
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
Cauerstr. 11
Erlangen, Germany
June 2017

Jürgen Teich

Contents

Volume 1

Volume 2

About the Editors

Soonhoi Ha
Department of Computer Science and Engineering
Seoul National University
Gwanak-ro 1, Gwanak-gu
Seoul, Korea

Soonhoi Ha received the B.S. and M.S. degrees in Electronics Engineering from Seoul National University, Seoul, Korea, in 1985 and 1987, respectively, and the Ph.D. degree in Electrical Engineering and Computer Science from the University of California at Berkeley, Berkeley, CA, USA, in 1992. He is currently a professor with Seoul National University. His current research interests include HW/SW codesign of embedded systems, system simulation, and robust embedded software design. Prof. Ha has actively participated in the premier international conferences in the EDA area, serving CODES+ISSS 2006, ASP-DAC 2008, and ESTIMedia 2005–2006 as the program cochair and ESWeek 2017 as the vice general chair. He is an IEEE Fellow and a member of ACM.

Jürgen Teich
Department of Computer Science
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
Cauerstr. 11
Erlangen, Germany

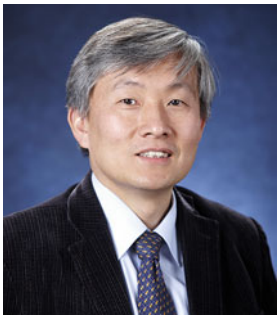
Jürgen Teich is with Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany, where he is directing the Chair for Hardware/Software Codesign since 2003. He received the M.S. degree (Dipl.- Ing.; with honors) from the University of Kaiserslautern, Germany, in 1989 and the Ph.D. degree (Dr.-Ing.; summa cum laude) from the University of Saarland, Saarbruecken, Germany, in 1993.

Prof. Teich has organized various ACM/IEEE conferences/symposia as program chair including CODES+ISSS 2007, FPL 2008, ASAP 2010, and DATE 2016. He also serves as the vice general chair of DATE 2018 as well as in the editorial board of scientific journals including *ACM TODAES*, *IEEE Design and Test*, and *JES*. He has edited two textbooks on hardware/software codesign (Springer).

Since 2010, he has also been the principal coordinator of the Transregional Research Center 89 "Invasive Computing" on multicore research funded by the German Research Foundation (DFG). Since 2011, he is a member of Academia Europaea, the Academy of Europe.

Section Editors

Part I: Introduction to Hardware/Software Codesign

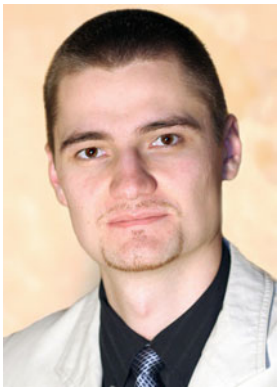


Soonhoi Ha Department of Computer Science and Engineering, Seoul National University, Gwanak-gu, Seoul, Korea
sha@snu.ac.kr

Part II: Models and Languages for Codesign

Christian Haubelt Institute of Applied Microelectronics and Computer Engineering, University of Rostock, Rostock, Germany
christian.haubelt@uni-rostock.de

Part III: Design Space Exploration



Michael Glass Institute of Embedded Systems/Real-Time Systems at Ulm University, Ulm, Germany
michael.glass@uni-ulm.de

Part IV: Processor, Memory, and Communication Architecture Design

Tulika Mitra Department of Computer Science, School of Computing, National University of Singapore, Singapore, Singapore
tulika@comp.nus.edu.sg

Part V: Hardware/Software Cosimulation and Prototyping

Rainer Dömer Center for Embedded and Cyber-physical Systems, Department of Electrical Engineering and Computer Science, The Henry Samueli School of Engineering, University of California at Irvine, Irvine, CA, USA
doemer@uci.edu

Part VI: Performance Estimation, Analysis, and Verification

Petru Eles Department of Computer and Information Science, Linkoping University, Linköping, Sweden
petru.eles@liu.se

Part VII: Hardware/Software Compilation and Synthesis

Aviral Shrivastava School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, USA
aviral.shrivastava@asu.edu

Part VIII: Codesign Tools and Environment

Andreas Gerstlauer Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA
gerstl@ece.utexas.edu

Part IX: Applications and Case Studies

Shuvra Bhattacharyya Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, USA
Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland
ssb@umd.edu

Contributors

Miguel Angel Aguilar Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Aachen, Germany

Leonie Ahrendts Institute of Computer and Network Engineering, Technical University Braunschweig, Braunschweig, Germany

Benny Akesson Eindhoven University of Technology, Eindhoven, The Netherlands

David Atienza Embedded Systems Laboratory (ESL), EPFL, Lausanne, Switzerland

Seyed-Hosein Attarzadeh-Niaki Shahid Beheshti University (SBU), Tehran, Iran

Philip Axer NXP Semiconductors, Hamburg, Germany

Hadi Ahmadi Balef Eindhoven University of Technology, Eindhoven, The Netherlands

Giovanni Beltrame Polytechnique Montréal, Montreal, QC, Canada

Shuvra S. Bhattacharyya Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD, USA

Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland

Haseeb Bokhari University of New South Wales (UNSW), Sydney, NSW, Australia

Jens Brandt Faculty of Electrical Engineering and Computer Science, Hochschule Niederrhein, Krefeld, Germany

Gabriela Breaban Eindhoven University of Technology, Eindhoven, The Netherlands

Oliver Bringmann Wilhelm-Schickard-Institut, University of Tübingen, Tübingen, Germany

Embedded Systems, University of Tübingen, Tübingen, Germany

Davide Brunelli Department of Industrial Engineering, University of Trento, Trento, Italy

Jian Cai Arizona State University, Tempe, AZ, USA

Jeronimo Castrillon Center for Advancing Electronics Dresden, TU Dresden, Dresden, Germany

Samarjit Chakraborty TU Munich, Munich, Germany

Wanli Chang Singapore Institute of Technology, Singapore, Singapore

Jae-Min Cho Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea

Kiyoung Choi Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea

Pablo G. Del Valle Embedded Systems Laboratory (ESL), EPFL, Lausanne, Switzerland

Clément Deschamps Antfield SAS, Grenoble, France

Jonas Diemer Symtavision, Braunschweig, Germany

Rainer Dömer Center for Embedded and Cyber-Physical Systems, Department of Electrical Engineering and Computer Science, The Henry Samueli School of Engineering, University of California, Irvine, CA, USA

Nikil Dutt Center for Embedded and Cyber-Physical Systems, University of California Irvine, Irvine, CA, USA

Wolfgang Ecker Infineon Technologies AG, Neubiberg, Germany

Petru Eles Department of Computer and Information Science, Linköping University, Linköping, Sweden

Ahmed Eltawil Center for Embedded and Cyber-Physical Systems, University of California Irvine, Irvine, CA, USA

Rolf Ernst Institute of Computer and Network Engineering, Technical University Braunschweig, Braunschweig, Germany

Juan Fernando Eusse Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Aachen, Germany

Heiko Falk Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany

Joachim Falk Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany

Franco Fummi Università di Verona, Verona, Italy

Marc Geilen Eindhoven University of Technology, Eindhoven, The Netherlands

Andreas Gerstlauer Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA

Christoph Gerum Embedded Systems, University of Tübingen, Tübingen, Germany

Michael Glaß Institute of Embedded Systems/Real-Time Systems at Ulm University, Ulm, Germany

Kees Goossens Eindhoven University of Technology, Eindhoven, The Netherlands

Sven Goossens Eindhoven University of Technology, Eindhoven, The Netherlands

Dip Goswami Eindhoven University of Technology, Eindhoven, The Netherlands

Soonhoi Ha Department of Computer Science and Engineering, Seoul National University, Gwanak-gu, Seoul, Korea

Christian Haubelt Department of Computer Science and Electrical Engineering, Institute of Applied Microelectronics and Computer Engineering, University of Rostock, Rostock, Germany

Jörg Henkel Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Fernando Herrera GESE Group, TEISA Department, ETSIIT, Universidad de Cantabria, Santander, Cantabria, Spain

Robin Hofmann Institute of Computer and Network Engineering, Technical University Braunschweig, Braunschweig, Germany

Pekka Jääskeläinen Tampere University of Technology, Tampere, Finland

Axel Jantsch Vienna University of Technology, Vienna, Austria

Prachi Joshi Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA

Hanwoong Jung Seoul National University, Gwanak-gu, Seoul, Korea

Amin Khajeh Broadcom Corp., San Jose, CA, USA

Martijn Koedam Eindhoven University of Technology, Eindhoven, The Netherlands

Tim Kogel Synopsys, Inc., Aachen, Germany

Fadi Kurdahi Center for Embedded and Cyber-Physical Systems, University of California Irvine, Irvine, CA, USA

Luciano Lavagno Politecnico di Torino, Torino, Italy

Mihai Teodor Lazarescu Politecnico di Torino, Torino, Italy

Kyunghun Lee Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA

Rainer Leupers Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Aachen, Germany

Lin Li Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA

Yonghui Li Eindhoven University of Technology, Eindhoven, The Netherlands

Shuoxin Lin Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA

Guantao Liu Center for Embedded and Cyber-Physical Systems, University of California, Irvine, CA, USA

Yanzhou Liu Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA

Martin Lukasiewycz Robert Bosch GmbH, Corporate Research, Renningen, Germany

Grant Martin Cadence Design Systems, San Jose, CA, USA

Peter Marwedel Computer Science, TU Dortmund University, Dortmund, Germany

Julio Medina Software Engineering and Real-Time Group, University of Cantabria, Santander, Cantabria, Spain

Luc Michel Antfield SAS, Grenoble, France

Tulika Mitra Department of Computer Science, School of Computing, National University of Singapore, Singapore, Singapore

Daniel Mueller-Gritschneder Department of Electrical and Computer Engineering, Technical University of Munich, Munich, Germany

Majid Nabi Eindhoven University of Technology, Eindhoven, The Netherlands

Andrew Nelson Eindhoven University of Technology, Eindhoven, The Netherlands

Olaf Neugebauer Computer Science, TU Dortmund University, Dortmund, Germany

Hristo Nikolov Leiden University, Leiden, The Netherlands

Sebastian Ottlik Microelectronic System Design, FZI Research Center for Information Technology, Karlsruhe, Germany

Santiago Pagani Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Ali Pahlevan Embedded Systems Laboratory (ESL), EPFL, Lausanne, Switzerland

Preeti Ranjan Panda Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi, India

Jacopo Panerati Polytechnique Montréal, Montreal, QC, Canada

Sri Parameswaran University of New South Wales (UNSW), Sydney, NSW, Australia

Young-Hwan Park Digital Media and Communications R&D Center, Samsung Electronics, Seoul, Korea

Frédéric Pétrot Université de Grenoble Alpes, Grenoble, France

Andy Pimentel University of Amsterdam, Amsterdam, The Netherlands

Teemu Pitkänen Ajat Oy, Espoo, Finland

William Plishker Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA

Graziano Pravadelli Università di Verona, Verona, Italy

Davide Quaglia Università di Verona, Verona, Italy

Felix Reimann Audi Electronics Venture GmbH, Gaimersheim, Germany

Maurizio Rossi Department of Industrial Engineering, University of Trento, Trento, Italy

Debayan Roy TU Munich, Munich, Germany

Ingo Sander KTH Royal Institute of Technology, Stockholm, Sweden

Santanu Sarma University of California Irvine, Irvine, CA, USA

Gunar Schirner Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA

Frank Schirrmeister Cadence Design Systems, San Jose, CA, USA

Tim Schmidt Center for Embedded and Cyber-Physical Systems, University of California, Irvine, CA, USA

Klaus Schneider Embedded Systems Group, University of Kaiserslautern, Kaiserslautern, Germany

Johannes Schreiner Infineon Technologies AG, Neubiberg, Germany

Donatella Sciuto Politecnico di Milano, Milano, Italy

Muhammad Shafique Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Mansureh Shahraki Moghaddam Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea

Weihua Sheng Silexica GmbH, Köln, Germany

Jun Yong Shin Center for Embedded and Cyber-Physical Systems, University of California Irvine, Irvine, CA, USA

Aviral Shrivastava School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA

Shubhendu Sinha Eindhoven University of Technology, Eindhoven, The Netherlands

Todor Stefanov Leiden University, Leiden, The Netherlands

Sander Stuijk Eindhoven University of Technology, Eindhoven, The Netherlands

Jarmo Takala Tampere University of Technology, Tampere, Finland

Rasool Tavakoli Eindhoven University of Technology, Eindhoven, The Netherlands

Jürgen Teich Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany

Daniel Thiele Elektrobit Automotive GmbH, Erlangen, Germany

Juan Valencia Eindhoven University of Technology, Eindhoven, The Netherlands

Reinier van Kampenhout Eindhoven University of Technology, Eindhoven, The Netherlands

Peter van Stralen Philips Healthcare, Best, The Netherlands

Alexander Viehl Microelectronic System Design, FZI Research Center for Information Technology, Karlsruhe, Germany

Eugenio Villar GESE Group, TEISA Department, ETSIIT, Universidad de Cantabria, Santander, Cantabria, Spain

Sara Vinco Politecnico di Torino, Turin, Italy

Yosinori Watanabe Cadence Design Systems, San Jose, CA, USA

Marilyn Wolf School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA

Chun Jason Xue City University of Hong Kong, Hong Kong, Hong Kong

Christian Zebelein Valeo Siemens eAutomotive Germany GmbH, Erlangen, Germany

Haibo Zeng Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA

Licong Zhang TU Munich, Munich, Germany

List of Acronyms

6LoWPAN	IPv6 over Low Power Wireless Personal Area Network
ABV	Assertion-Based Verification
AC	Alternating Current
ACK	Acknowledgement
ADAS	Advanced Driver Assistance System
ADC	Analog-to-Digital Converter
ADF	Architecture Description File
ADG	Approximated Dependence Graph
ADL	Architecture Description Language
ADM	Abstract Design Module
ADRS	Average Distance from Reference Set
ADT	Abstract Data Type
AFDX	Avionics Full-Duplex Switched Ethernet
AHB	Advanced High-performance Bus
AIF	Averest Intermediate Format
ALAP	As Late As Possible
ALM	Adaptive Logic Module
ALU	Arithmetic-Logic Unit
ANN	Artificial Neural Network
APB	Advanced Peripheral Bus
API	Application Programming Interface
ARM	Advanced Risc Machines
ARQ	Automatic Repeat Request
ASAP	As Soon as Possible
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
ASK	Amplitude Shift Key
ASMBL	Advanced Silicon Modular Block
ASP	Application-Specific Processor
AST	Abstract Syntax Tree
AT	Approximately Timed

AT-BP Approximately Timed Base Protocol
AV Architects View
AVB Audio/Video Bridging
AXI Advanced eXtensible Interface
BB Basic Block
BCET Best-Case Execution Time
BCRT Best-Case Response Time
BD Budget Descriptor
BDF Boolean Data Flow
BER Bit Error Rate
BERET Bundled Execution of REcurring Traces
BFD Best-Fit-Decreasing
BFM Bus-Functional Model
BIST Built-In Self-Test
BLB Bit Lock Block
BLM Block-Level Model
BLS Binary-Level Simulation
BNF Backus-Naur Form
BOM Bill of Materials
BPSK Binary PSK
BRF Bypass Register File
BSP Board Support Package
BTB Branch Target Buffer
CA Cycle Accurate
CAD Computer-Aided Design
CAL Cal Actor Language
CAN Controller Area Network
CCA Configurable Compute Accelerator
CC Communication Controller
CCE Configuration Cache Element
CCSP Credit-Controlled Static Priority
CDC Clock Domain Crossing
CDFG Control-/Data-Flow Graph
CDMA Code Division Multiple Access
CE Communication Element
CFDF Core Functional Data Flow
CFG Control-Flow Graph
CFU Custom Functional Unit
CGA Coarse-Grained Array
CG Call Graph
CGRA Coarse Grained Reconfigurable Architecture
CIC Common Intermediate Code
CIL Compiler-In-the-Loop
CIM Computation Independent Model
CIS Custom Instruction-Set

CLB	Configurable Logic Block
CLDSE	Cross-Layer Design Space Exploration
CM	Communication Memory
CMOS	Complementary Metal-Oxide-Semiconductor
CMP	Chip Multi-Processor
CNN	Convolutional Neural Network
CORDIC	COordinate Rotational DIgital Computer
CVMP	Correlation-aware VM Placement
COTS	Commercial/Components Off-The-Shelf
CPA	Compositional Performance Analysis
CPF	Common Power Format
CPN	C for Process Networks
CPS	Cyber-Physical System
CPU	Central Processing Unit
CRAC	Computer Room Air Conditioning
CRC	Cyclic Redundancy Check
CRPD	Cache-Related Preemption Delay
CSDF	Cyclo-Static Data Flow
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
CTI	Charge Transfer Interconnect
CTL	Computation Tree Logic
CUDA	Compute Unified Device Architecture
CV	Computer Vision
D2H	Device-to-Host
DAG	Directed Acyclic Graph
DB	Database
DBT	Dynamic Binary Translation
DC	Direct Current
DCG	Dynamic Call Graph
DCT	Discrete Cosine Transform
DDF	Dennis Data Flow
DDR	Double Data Rate
DE	Discrete Event
DES	Discrete Event Simulation
DFG	Data-Flow Graph/Dependence-Flow Graph
DFT	Discrete Fourier Transfrom
DICE	DSPCAD Integrative Command Line Environment
DIF	Decimation-in-Frequency/Data-flow Interchange Format
DISC	Dynamic Instruction-Set Computer
DIT	Decimation-in-Time
DLMB	Data Local Memory Bus
DLP	Data-Level Parallelism
DMA	Direct Memory Access
DMAMEM	DMA Memory
DMEM	Data Memory

DMI Direct Memory Interface
DoD Depth-of-Discharge
DoE Design of Experiments
DOR Dimension Ordered Routing
DP Dynamic Programming
DPLL Davis-Putnam-Logemann-Loveland
DPM Dynamic Power Management
DPR Dynamic Partial Reconfiguration
DRAA Dynamically Reconfigurable ALU Array
DRAM Dynamic Random-Access Memory
DRESC Dynamically Reconfigurable Embedded System Compiler
DSE Design Space Exploration
DSL Domain-Specific Language
DSML Domain-Specific Modeling Language
DSO Distribution System Operator
DSP Digital Signal Processor/Digital Signal Processing
DTA Dynamic Timing Analysis
DTM Dynamic Thermal Management
DUT Design Under Test
DUV Design Under Verification
DVFS Dynamic Voltage and Frequency Scaling
DVS Dynamic Voltage Scaling
DWM Domain Wall Memory
DWT Discrete Wavelet Transform
EA Evolutionary Algorithm
EBNF Extended Backus-Naur Form
ECO Engineering Change Order
ECU Electronic Control Unit
EDA Electronic Design Automation
EDF Earliest Deadline First
EDP Energy-Delay Product
EDSP Energy-Delay Square Product
E/E Electric and Electronic
EEPROM Electrically Erasable Programmable Read-Only Memory
EFSM Extended Finite-State Machine
EGRA Expression Grained Reconfigurable Array
ELF Executable and Linkable Format
EMB Electro-Mechanical Brake
EMF Eclipse Modeling-Framework
EML Execution Modeling Level
EMS Edge Centric Modulo Scheduling
EOH Extremal Optimization meta-Heuristic
ES Embedded System
ESL Electronic System Level
ESS Energy Storage Systems

ET	Event-Triggered/Execution Time
ETSCH	Extended TSCH
EWFD	Equally-Worst-Fit-Decreasing
FBSP	Frame-Based Static Priority
FCFS	First-Come First-Serve
FDS	Force-Directed Scheduling
FeRAM	Ferro-electric Random-Access Memory
FFT	Fast Fourier Transform
FIFO	First-In First-Out
FIR	Finite Impulse Response
ForSyDe	Formal System Design
FPGA	Field-Programmable Gate Array
FS	Feature Selection
FSM	Finite State Machine
FTDMA	Flexible Time Division Multiple Access
FT	Fast Timed
FunState	Functions Driven by State Machines
GA	Genetic Algorithm
GALS	Globally Asynchronous Locally Synchronous
GCC	GNU Compiler Collection
GFRBM	Generic File Reader Bus Master
GIPS	Giga-Instruction Per Second
GLV	Graph-Level Vectorization
GME	Generic Modeling Environment
GOPS	Giga Operations Per Second
GPGPU	General-Purpose computing on Graphics Processing Units
GPIO	General-Purpose Input/Output-pin
GPP	General-Purpose Processor
GPRS	General Packet Radio Service
GPT	General-Purpose Timer
GPU	Graphics Processing Unit
GUI	Graphical User Interface
H2D	Host-to-Device
HAL	Hardware Abstraction Layer
HAPS	High-performance ASIC Prototyping System
HDB	Hardware Database
HDL	Hardware Description Language
HDS	Hardware-Dependent Software
HES	Hybrid Electric Systems
HLS	High-Level Synthesis
HMP	Heterogeneous Multi-core Processor
HPC	Horizontally Partitioned Cache
HRM	Hardware Resource Modeling
HSCD	Hardware/Software Codesign
HSDF	Homogeneous (Synchronous) Data Flow

HTML Hypertext Markup Language
HVL Hardware Verification Language
HW Hardware
I2C Inter-Integrated Circuit
ICFG Interprocedural Control-Flow Graph
ICT Information and Communications Technology
ICU Input Capture Unit
IDC Inquisitive Defect Cache
IDE Integrated Development Environment
ID Identifier
IEEE Institute of Electrical and Electronics Engineers
II Initiation Interval
ILMB Instruction Local Memory Bus
ILP Integer Linear Program/Instruction-Level Parallelism
IMEM Instruction Memory
IMS Iterative Modulo Scheduling
IOE I/O Element
I/O Input/Output
IoT Internet of Things
IPC Inter-Process Communication/Instructions Per Cycle
IP Intellectual Property
IPB Intellectual Property Block
IPM Intellectual Property Module
IPS Instruction Per Second
IR Intermediate Representation
ISA Instruction-Set Architecture
ISEF Stretch Instruction-Set Extension Fabric
ISR Interrupt Service Routine
ISS Instruction-Set Simulator
IT Information Technology
ITRS International Technology Roadmap for Semiconductors
ITS Individual Test Subdirectory
JPEG Joint Photographic Experts Group
JSON JavaScript Object Notation
JTAG Joint Test Action Group
KPN Kahn Process Network
LAB Logic Array Block
LCS Live Cache States
LE Logic Element
LIDE LIghtweight Data-flow Environment
LIN Local Interconnect Network
LISA Language for Instruction-Set Architectures
LLVM Low-Level Virtual Machine
LRU Least-Recently Used
LS List Scheduling

LTF Largest Task First
LTL Linear Time Logic
LT Loosely Timed
LUT Look-Up Table
M2M Model-to-Model
MAC Media Access Control/Multiply-Accumulator
MAPE Mean Average Percentage Error
MAPS MPSoC Application Programming Studio
MARTE Modeling and Analysis of Real-Time Embedded Systems
MBD Model-Based Design
MCO Multi-Core Optimization
MCR Mode Change Request
MCS Mixed-Criticality System
MDA Model-Driven Architecture
MDD Model-Driven Design
MDP Markov Decision Process
MDSDF Multi-Dimensional Synchronous Data Flow
MILP Mixed Integer Linear Programming
MIMO Multiple Input Multiple Output
MIPS Million Instructions Per Second
MIR Medical Image Registration
MISO Multiple Input Single Output
MJPEG Motion JPEG
MLBJ Multi-Level Back Jumping
MLoC Million Lines of Code
MMC/SD Multimedia/Secure Digital Card
MMIO Memory-Mapped I/O
MMU Memory Management Unit
MoC Model of Computation
MOF Meta Object Facility
MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor
MOST Media Oriented Systems Transport
MPSoC Multi-Processor System-on-Chip
MRRG Modulo Resource Routing Graph
MTF Mean Time to Failure
MTJ Magnetic Tunnel Junction
MTM Mode Transition Machine
NDF Non-Determinate Data Flow
NFP Non-Functional Property
NI Network Interface
nML not a Machine Language
NMOS Negative-type Metal-Oxide-Semiconductor
NN Neural Network
NoC Network-on-Chip
NOCT Nominal Operating Cell Temperature

NRE Non-Recurring Engineering
NVIC Nested Vectored Interrupt Controller
NVM Non-Volatile Memory
OCL Object Constraint Language
OFDM Orthogonal Frequency Dependent Multiplexing
OMG Object Management Group
OOO PDES Out-of-Order Parallel Discrete Event Simulation
OSAL Operation Set Abstraction Layer
OSCI Open SystemC Initiative
OS Operating System
OT Operation Table
OVL Open Verification Library
OVM Open Verification Methodology
PAMONO Plasmon-Assisted Microscopy of Nano-Objects
PB Pseudo Boolean
PCI Peripheral Component Interconnect
PCM Phase Change Memory
PC Personal Computer
PCP Peak Clustering-based Placement
PDES Parallel Discrete Event Simulation
PDF Probability Density Function
PDU Power Distribution Unit
PE Processing Element
PFU Programmable Functional Unit
PI Principal Investigator
PIC Programmable Interrupt Controller
PIM Platform Independent Model
PIP Parametric Integer Programming
PLB Processor Local Bus
PLL Phase Locked Loop
PLP Pipeline-Level Parallelism
PMOS Positive-type Metal-Oxide-Semiconductor
PMU Power Management Unit
PNG Portable Network Graphics
PN Process Network
PPN Polyhedral Process Network
PREESM Parallel and Real-time Embedded Executives Scheduling Method
PRISC Programmable Instruction-Set Processor
PSDF Parameterized Synchronous Data Flow
PSK Phase Shift Keying
PSL Property Specification Language
PSM Program State Machine/Parameterized Sets of Modes/Platform Specific Model
PSNR Peak SNR
PSO Particle Swarm Optimization

PSTC Path Segment Timing Characterization
PV Photovoltaic
PVT Programmers View Time
PWM Pulse-Width Modulation
QAM Quadrature Amplitude Modulation
QEA Quantum-inspired Evolutionary Algorithm
QoS Quality of Service
QPSK Quadrature PSK
RAM Random-Access Memory
RAW Read-After-Write
RCM Reconfigurable Computing Module
RC Resistor-Capacitor/Reconfigurable Cell
RCS Reaching Cache States
RDF Random Dopant Fluctuations
RFID Radio-Frequency Identification
RF Register File/Radio Frequency
RFTS Run Fast Then Stop
RISC Reduced Instruction-Set Processor/Recoding Infrastructure for SystemC
RISPP Rotating Instruction-Set Processing Platform
RLD Run Length Decoding
ROM Read-Only Memory
RR Round Robin
RRAM Resistive Random-Access Memory
RSM Response Surface Modeling
RST ReServation Table
RT Response Time
RTC Real-Time Clock
RTL Register Transfer Level
RTOS Real-Time Operating System
RVC Reconfigurable Video Coding
SADF Scenario-Aware Data Flow
SANLP Static Affine Nested Loop Program
SA Simulated Annealing
SAT Boolean Satisfiability
SBS Sequential Backward Selection
SCC Single Chip Cloud computer/Strongly Connected Component
SCE System-on-Chip Environment
SCML SystemC Modeling Library
SDC Secure Digital Card
SDF Synchronous Data Flow
SDK Software Development Kit
SDR Software Defined Radio
SDS System Development Suite
SDTC Scheduling and Data Transfer Configuration

SERE Sequential Extended Regular Expression
SESE Single-Entry Single-Exit
SFA Single Frequency Approximation
SFS Sequential Forward Selection
SFU Specialized Functional Unit
SG Segment Graph
SI Scheduling Interval
SIMD Single Instruction, Multiple Data
SIMT Single Instruction, Multiple Threads
SLDL System-Level Description Language
SLD System-Level Design
SLP System-Level Power
SLS Source-Level Simulation/System-Level Synthesis
SMP Symmetric Multi-Processing
SMT Satisfiability Modulo Theories
SMV Symbolic Model Verifier
SNR Signal-to-Noise Ratio
SoC System-on-Chip/State of Charge
SoH State of Health
SPI Serial Peripheral Interface/Signal Passing Interface
SPKM Split & Push Kernel Mapping
SPMD Single Program, Multiple Data
SPM Scratchpad Memory
SPNP Static-Priority Non-Preemptive
SPP Static Priority Preemptive
SPU Synergistic Processor Unit
SRAM Static Random-Access Memory
SSA Static Single Assignment
SSTA Statistical Static Timing Analysis
STC Standard Test Conditions
STMD Single Thread, Multiple Data
STree Schedule Tree
STT-RAM Spin-Transfer Torque Random-Access Memory
SVA System Verilog Assertions
SVM Support Vector Machine
SWC Software Cache
SW Software
SysteMoC SystemC Models of Computation
T-BCA Transaction-based Bus Cycle Accurate
TB Translation Block
TCE TTA-based Codesign Environment
TCL Tool Command Language
TCP/IP Transmission Control Protocol/Internet Protocol
TDB Timing Database
TDM Time-Division Multiplexing

TDMA	Time-Division Multiple Access
TDP	Thermal Design Power
TD	Temporal Decoupling
TFT	Thin-Film Transistor
TIE	Tensilica Instruction Extension
TIFU	Timer, Interrupt, and Frequency Unit
TLM	Transaction-Level Model
TLP	Task-Level Parallelism/Thread-Level Parallelism
TRM	Trace Replay Module
TSCH	Time-Synchronised Channel Hopping
TSN	Time-Sensitive Networking
TSP	Thermal Safe Power
TTA	Transport-Triggered Architecture
TT-CAN	Time-Triggered CAN
TTEthernet	Time-Triggered Ethernet
TTP	Time-Triggered Protocol
TT	Time-Triggered
TWCA	Typical Worst-Case Analysis
TWCRT	Typical Worst-Case Response Time
TWI	Two Wire Interface
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modeling Language
UPF	Unified Power Format
UPS	Uninterruptible Power Supply
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
UTP	Universal Testing Profile
UVM	Universal Verification Methodology
VEP	Virtual Execution Platform
VFI	Voltage/Frequency Island
VF	Vectorization Factor
V/f	Voltage/Frequency
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VIVU	Virtual Inlining and Virtual Unrolling
VLIW	Very Long Instruction Word
VLSI	Very-Large-Scale Integration
VM	Virtual Machine
VOS	Voltage Over Scaling
VPU	Virtual Processing Unit
VP	Virtual Prototype
VSIA	Virtual Socket Interface Alliance
VSL	Value Specification Language
VSP	Virtual System Platform
WAR	Write-After-Read

WAW	Write-After-Write
WCC	WCET-aware C Compiler
WCDMA	Wideband CDMA
WCEC	Worst-Case Energy Consumption
WCEP	Worst-Case Execution Path
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time
WL	Word Line
WSDF	Windowed Synchronous Data Flow
WSDL	Web Service Definition Language
WSN	Wireless Sensor Network
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema
XSLT	Extensible Stylesheet Language Transformations
YML	Y-chart Modeling Language

Part I
Introduction to Hardware/Software Codesign

Introduction to Hardware/Software Codesign

1

Soonhoi Ha, Jürgen Teich, Christian Haubelt, Michael Glaß, Tulika Mitra, Rainer Dömer, Petru Eles, Aviral Shrivastava, Andreas Gerstlauer, and Shuvra S. Bhattacharyya

Abstract

Hardware/Software Codesign (HSCD) is an integral part of modern Electronic System Level (ESL) design flows. This chapter will review important aspects of hardware/software codesign flows, summarize the historical evolution of codesign techniques, and subsequently summarize each of its major branches

S. Ha (✉)
Department of Computer Science and Engineering, Seoul National University, Gwanak-gu, Seoul, Korea
e-mail: sha@snu.ac.kr

J. Teich
Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany
e-mail: juergen.teich@fau.de

C. Haubelt
Department of Computer Science and Electrical Engineering, Institute of Applied Microelectronics and Computer Engineering, University of Rostock, Rostock, Germany
e-mail: christian.haubelt@uni-rostock.de

M. Glaß
Institute of Embedded Systems/Real-Time Systems at Ulm University, Ulm, Germany
e-mail: michael.glass@uni-ulm.de

T. Mitra
Department of Computer Science, School of Computing, National University of Singapore, Singapore, Singapore
e-mail: tulika@comp.nus.edu.sg

R. Dömer
Center for Embedded and Cyber-Physical Systems, Department of Electrical Engineering and Computer Science, The Henry Samueli School of Engineering, University of California, Irvine, CA, USA
e-mail: doemer@uci.edu

P. Eles
Department of Computer and Information Science, Linköping University, Linköping, Sweden
e-mail: petru.eles@liu.se

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_41

of research and achievements that later will be presented in detail by different parts of this Handbook of Hardware/Software Codesign.

Acronyms

ASIC	Application-Specific Integrated Circuit
DES	Discrete Event Simulation
DSE	Design Space Exploration
EA	Evolutionary Algorithm
EDA	Electronic Design Automation
ESL	Electronic System Level
ForSyDe	Formal System Design
FSM	Finite-State Machine
GA	Genetic Algorithm
HSCD	Hardware/Software Codesign
HW	Hardware
ILP	Integer Linear Program
IP	Intellectual Property
ISA	Instruction-Set Architecture
KPN	Kahn Process Network
MARTE	Modeling and Analysis of Real-Time Embedded Systems
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
OOO PDES	Out-of-Order Parallel Discrete Event Simulation
OS	Operating System
PB	Pseudo-Boolean
PDES	Parallel Discrete Event Simulation
PN	Process Network
SDF	Synchronous Data Flow
SIMD	Single Instruction, Multiple Data
SLDL	System-Level Description Language

A. Shrivastava
School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA
e-mail: aviral.shrivastava@asu.edu

A. Gerstlauer
Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA
e-mail: gerstl@ece.utexas.edu

S.S. Bhattacharyya
Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD, USA

Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland
e-mail: ssb@umd.edu

SoC	System-on-Chip
SW	Software
SysteMoC	SystemC Models of Computation
TLM	Transaction-Level Model
UML	Unified Modeling Language
VLIW	Very Long Instruction Word
VP	Virtual Prototype
WCET	Worst-Case Execution Time

Contents

1.1 Introduction

As the name implies, Hardware/Software Codesign (HSCD) denotes design methodologies for electronic systems that exploit the trade-offs and the synergy of Hardware (HW) and Software (SW). Typically, the application functionality is partitioned into software components that are running on the processor cores and hardware components that are used to accelerate some parts of the application or to provide interfaces to the environment. In the traditional design practice for such systems, software is usually designed after the hardware architecture is fixed as illustrated in Fig. 1.1.

The design starts with a set of specifications and requirements on the functional and nonfunctional properties of the target system. The first step is to determine the appropriate algorithm to meet the functional specification if it is not given. Afterward, system architects typically determine a hardware architecture that would satisfy the functional requirements while optimizing the design objectives such as cost minimization and/or energy minimization. The decision is usually made based on the profiling information of the algorithms to be implemented and their computational complexity as well as resource requirements and constraints. It is typically influenced by past design experience of similar systems. How to partition the functionality into software and hardware components is also determined manually in this step.

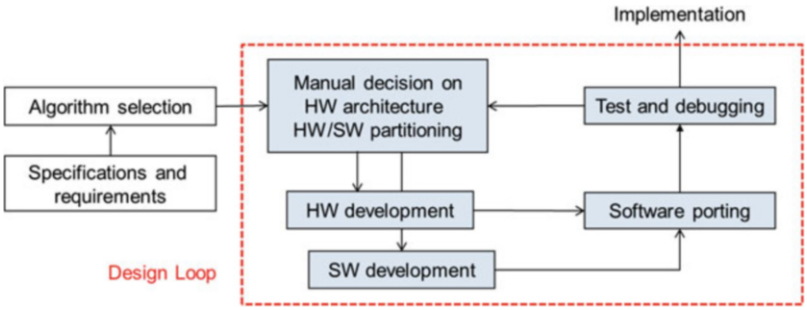


Fig. 1.1 A traditional design flow of an electronic system. The HW architecture including programmable cores and HW components is designed first. The SW development including tests starts only after HW development. Moreover, the decisions, which parts of the specification are realized in HW and which parts in SW, are done manually

After this HW/SW partitioning for the determined hardware architecture, two groups of engineers work independently to design and implement the hardware components and software components. The software group waits to finalize the design until a hardware prototype is made by the hardware group since the software usually contains hardware-dependent components. After the hardware prototype is built, the entire software that consists of the system software and the application software is ported. After porting is completed, the system undergoes testing and verification that checks whether the system satisfies the functional and nonfunctional requirements. It rarely happens that testing and verification succeeds at the first attempt. If a test fails, the causes of the failure need to be found, which is a difficult and tiresome task for all engineers involved in the design. If it turns out that the hardware architecture and HW/SW partitioning decisions need to be changed, it is required to go back to the starting point of the design loop to iterate the design process.

As the system complexity increases, the above traditional design flow faces several challenges. It is evident that the critical path in the design process becomes prohibitively long and costly in case multiple iterations of the sequential flow are performed from HW and SW development, software porting, and testing. And the design quality mostly depends on the expertise of the architect since the target architecture and HW/SW partitioning are decided manually.

HW/SW codesign overcomes those challenges in a systematic way as illustrated in Fig. 1.2. It shortens the critical path of the design loop by estimating the performance fast and accurately for a given hardware platform and HW/SW partitioning decision without hardware prototyping. In Fig. 1.3 taken from [31], it is shown that considerable amounts of design time may be saved from the concurrent design of hardware and software by codesign flows according to Fig. 1.2. A popular technique for performance estimation is HW/SW cosimulation in which the software is running on a virtual prototype of a given hardware architecture. Fast and accurate performance estimation enables us to explore the wide design space

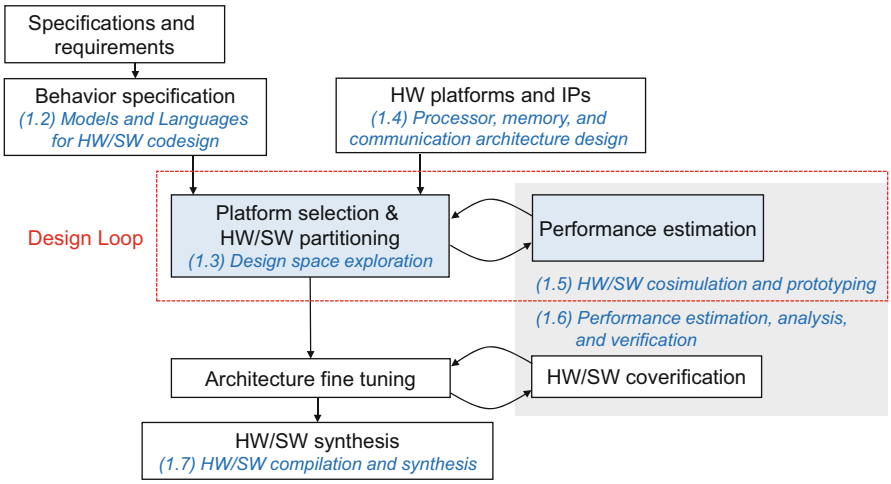


Fig. 1.2 Design flow based on HW/SW codesign

that is defined by several axes such as candidate HW architectures and HW/SW partitioning solutions. This step is called "Design Space Exploration (DSE)" and aims to explore a set of Pareto-optimal solutions for the system with various design objective functions. From the DSE step, a target HW architecture and the associated HW/SW partitioning decision is produced, which would be of better quality than the one made manually by experts in the traditional design method. Afterward, the architecture is fine-tuned, which determines the detailed microarchitecture of the system and allows to verify the system behavior more accurately. HW/SW cosimulation can be used also for HW/SW coverification with more detailed modeling of hardware components while sacrificing the simulation performance. Another popular method for HW/SW coverification is to use an emulation system. Final implementation is made after the correctness of the design is verified.

An important issue of this methodology is how to specify the system behavior or algorithm. It is believed that a good specification method should not be biased to any specific implementation. If a C language program is given as the input specification model for the HW/SW codesign methodology, the design space to explore will be severely restricted to a single processor system that has some hardware IPs to accelerate compute-intensive portions of the algorithm since partitioning a C program into multiple processors is not an easy task. Thus, the specification model should be easy to be partitioned into HW and SW components.

HW/SW codesign covers the full spectrum of system design issues from initial behavior specification to final implementation. In this introductory chapter to the Handbook of Hardware/Software Codesign, we give the overview of each design step explaining the research issues and the current status as well as the future perspective. Section 1.2 explains why various models of computation have been proposed for behavior specification of applications and introduces some

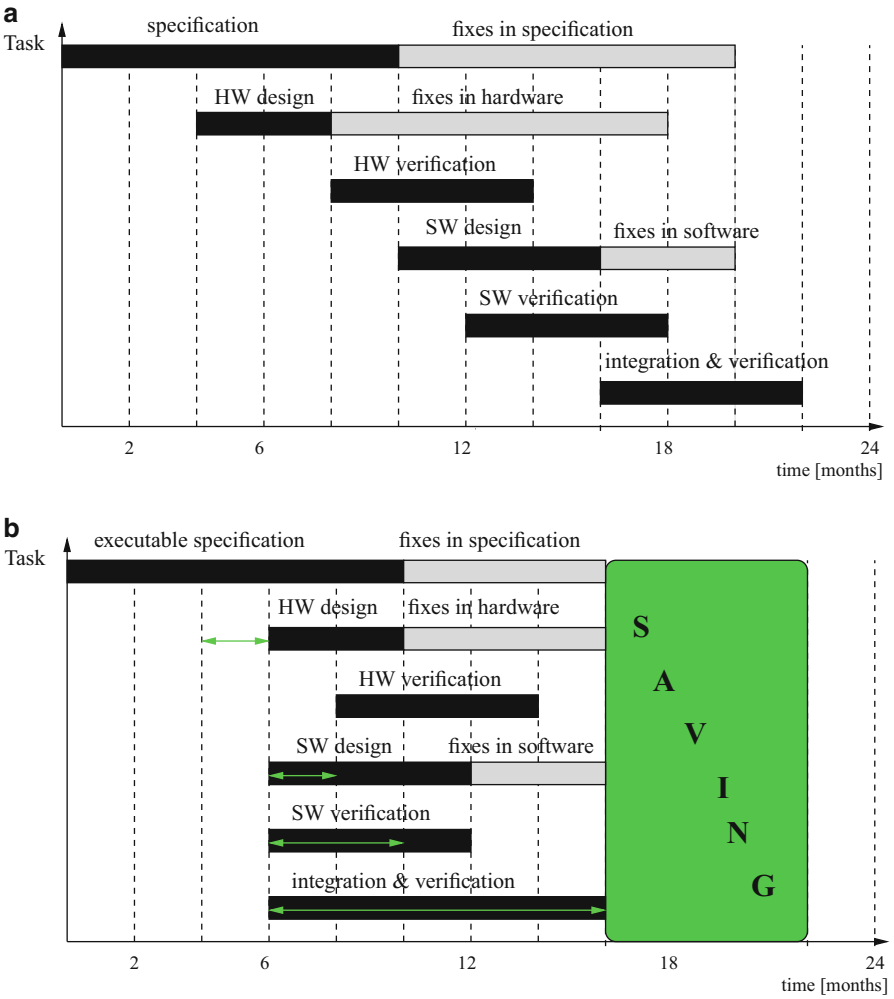


Fig. 1.3 Comparison of (**a**) a classical design flow and (**b**) an ESL design flow starting from an executable specification and allowing for concurrent development of hardware and software after an initial delay for specification and design space exploration. In this figure according to [31], savings of several months of design time are possible

representative models. Section 1.3 defines the design space exploration problem as an optimization problem and reviews the state-of-the-art techniques for different variants and refinements of this problem. The section also presents how the DSE problem can be extended to cross-layer optimization and dynamic resource management. Section 1.4 provides a glimpse into unique hardware resources available for HW/SW codesign such as reconfigurable architectures and application-specific processors. How the memory hierarchy and interconnection network can

be customized is also discussed in the section. After the hardware platform is determined and the application is mapped to the platform, it is necessary to validate the functional correctness and performance satisfaction before actual implementation through prototyping or cosimulation. Section 1.5 introduces several state-of-the-art approaches to cosimulation and prototyping. Since the coverage of cosimulation and prototyping is limited to the input test set, no guarantee can be made to satisfy the correctness of the design. To overcome this limitation, various formal approaches to analyze the design have been proposed, which is discussed in Sect. 1.6. Section 1.7 addresses the issues and related techniques for software and hardware implementation of a given application on the system architecture based on the mapping decision made in the DSE step. In Sect. 1.8, some codesign tools and environments are reviewed, demonstrating how the codesign flow illustrated in Fig. 1.2 can be realized in a unified framework. The application domain of HW/SW codesign methodologies is widening to various types of computing systems, as will be discussed in Sect. 1.9. Finally, we conclude the chapter with a brief summary.

1.2 Models and Languages for Codesign

In order to shorten the critical path of the design loop of HW/SW systems, the specification of the system behavior plays a key role (cf. Fig. 1.2). As a starting point of nearly all modern HW/SW codesign methodologies, behavioral models are used to describe the desired application. In contrast, implementation models reflect design decisions and describe the structure of the system. Behavioral models for HW/SW codesign methodologies are the foundation for correct-by-construction design transformations and optimization. These models should not be biased toward any specific implementation style (HW or SW).

Hardware/software systems are designed for different application domains. Hence, the modeling focus can change from system to system, e.g. *reactive* systems model their responses to external events, while *transformative* systems transform input streams of data into output streams. These requirements have led to a variety of behavioral models and modeling languages for HW/SW systems. In order to classify different models, Edwards et al. [15] define the notion of Model of Computation (MoC) on the concept of the *tagged-signal model* [26]. In the tagged-signal model, signals are defined as a set of events where an event is given by a pair of a value and a tag. If the set of tags is totally ordered, the MoC is said to be timed; if the set is only partially ordered, the MoC is said to be untimed. Processes transform input signals into output signals. Different MoCs are distinguished by three largely orthogonal aspects, namely, *sequential behavior*, *concurrency*, and *communication* [15]. A coarse-grain classification of MoCs is into *activity-based* models, i.e., models describing processes and their dependencies, and *state-based* models, i.e., states and state transitions are represented by the model. By choosing a MoC, special attention has to be drawn to the trade-off between *expressiveness* and *analyzability*. The *expressiveness* of a MoC determines which kind of application can be modeled. However, increasing the degree of expressiveness limits the degree of *analyzability*.

Important MoCs include *process and data-flow networks* [22,25], where concurrent processes communicate via unbounded first-in-first-out queues; *communicating sequential processes* [21], where concurrent processes use rendezvous channels to synchronize; *discrete event models*, where events carry totally ordered time stamps [9]; and *synchronous/reactive models* [7], based on the perfect synchrony hypothesis, which assumes that stimuli and responses are simultaneous and that information is instantaneously broadcast. As the complexity of applications is still increasing, their control and data flow is tightly coupled, and the expressiveness of pure reactive or pure transformative models is no longer sufficient. As a consequence, new heterogeneous MoCs are required. A control/data-flow graph is presented in [17]. In [4], *codesign finite-state machines* are introduced as asynchronously communicating finite-state machines. A first systematic approach to combine different MoCs called **charts* has been presented in [19]. The *chart approach combines hierarchical pure (events without value), reactive, deterministic FSMs with concurrent MoCs (data flow, discrete event, synchronous/reactive, etc.).

Since then, many different modeling approaches have been proposed. Part 2, "Models and Languages for Codesign" of this handbook, *Models and Languages for Codesign*, gives a unique introduction to important modeling approaches based on well-defined MoCs. In four chapters, different MoCs and concepts of models and languages for codesign are discussed.

In ▸Chap. 2, "Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems", the *imperative* synchronous language *Quartz* is presented, which is based on the *synchronous MoC*. Since the synchronous MoC is also used to describe synchronous hardware circuits, a direct path to generating hardware from Quartz models exists. However, as synchronous MoCs are implementation independent, also software, and hardware/software systems, can be synthesized from Quartz models. In this chapter, the syntax and semantics of Quartz are described, and its analysis and synthesis based on *synchronous guarded actions* are presented.

A modeling approach based on a *data-flow MoC* is presented in ▸Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign". With their focus on concurrency, data-flow models are successfully applied in the modeling of hardware/software systems. Presenting the SystemC Models of Computation (SysteMoC) modeling approach, this chapter discusses the *expressiveness* and *analyzability* of different data-flow MoCs. Moreover, an automatic *classification* to detect the least expressive but most analyzable MoC within a *SysteMoC* model is described. In order to support the integration of SysteMoC in industrial design flows, an implementation based on the system description language SystemC exists.

In ▸Chap. 4, "ForSyDe: System Design Using a Functional Language and Models of Computation", the modeling approach *Formal System Design (ForSyDe)* is presented. The ForSyDe methodology combines the *functional programming* paradigm with the theory of models of computation. One key aspect of ForSyDe is its ability to model *heterogeneous* systems by supporting different MoCs and corresponding *MoC interfaces*. ForSyDe is language independent. This is shown by presenting a Haskell version and a SystemC version of ForSyDe.

Finally, ▸Chap. 5, "Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design Approach", presents a *graphical modeling* approach based on *UML/MARTE*. The Unified Modeling Language (UML) is a standard formalism for capturing system models in a graphical way. An important extension to UML is Modeling and Analysis of Real-Time Embedded Systems (MARTE), a UML profile providing syntactical and semantical extensions for the modeling and HW/SW real-time and embedded systems. Besides introducing the UML/MARTE modeling approach, this chapter also presents a modeling methodology, which addresses analysis and design activities and supports *single-source designs*.

1.3 Design Space Exploration

Already in early works, for example [27], the problem of concurrently defining a target platform, binding tasks to processors, and scheduling tasks has emerged. All the different possibilities arising from these steps are called the *design space* of the system. Different variants and refinements of this problem exist and, especially when also including hardware components, have become one of the essential design steps in HW/SW codesign. From the beginning, designers were not only interested in solving this problem to create systems that work properly but to create high-quality systems, inherently seeing the mentioned design steps as part of an *optimization problem*. Hence, to solve this optimization problem, the design space has to be *explored*.

Particularly in the context of HW/SW codesign, many initial works formulate this problem as a *bipartition problem*. There, each task is either assigned to a processor to be executed as software or to be implemented in hardware by a dedicated hardware accelerator. The typical assumption is that the execution of a task on a processor comes at an increased execution time and power consumption compared to a dedicated hardware accelerator. On the other hand, binding a task to the processor decreases the required amount of chip area and, hence, the cost. At this point, one can already recognize that it is not trivial to define what makes a high-quality system: One has to consider several different *quality numbers* (cost, area, execution time, power consumption, etc.) that are typically *conflicting*, i.e., an enhancement of one quality number very often comes at a deterioration of another. One possible solution is to weight each quality number according to the designer's needs and optimize the system regarding the sum of weighted quality numbers. The drawback is that only one system implementation will turn out to be *optimal* according to the predefined weights. Even worse, the designer has no insight on the various *trade-offs* that exist for the system under design. To overcome this, it has become state of the art for DSE to be formulated as a *multi-objective optimization problem*. There, each quality number is considered an individual design objective that has to be optimized. The goal of multi-objective optimization is to deliver not a single optimal but a set of so-called *Pareto-optimal* implementations, each being an optimal trade-off of all design objectives.

The ever-increasing complexity of the systems under design typically renders exhaustive searches for the best system implementations infeasible. Thus, the question which optimization techniques to apply for design space exploration arises. Allocating components, binding tasks, performing task scheduling and data routing, and often also setting crucial system parameters have, in almost all cases, the character of a combinatorial optimization problem. Moreover, many quality numbers cannot be reasonably determined with simple linear models but require complex calculations and/or even time-consuming simulations. These two aspects gave rise to meta-heuristics being applied to the DSE problem; see, for example [32], instead of solving the problem using exact approaches such as solving an Integer Linear Program (ILP).

In this handbook, ▸ Chap. 6, "Optimization Strategies in Design Space Exploration", introduces state-of-the-art multi-objective optimization techniques applied to HW/SW codesign. The chapter divides the techniques into four categories and compares them regarding crucial metrics such as convergence rate, scalability, and also in particular the setup effort for the designer. Also, results from extensive benchmarking are used to recommend techniques with respect to different properties of the design space. Another significant challenge for DSE is that the problem of allocation, binding, and scheduling itself is already an NP-complete problem. For complex systems, it may already be an almost unsolvable task for meta-heuristic optimization techniques to create *a single feasible implementation*. In such situations, the whole DSE is rendered infeasible. ▸ Chapter 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration", presents several hybrid optimization techniques that explicitly target such complex systems. The central idea is to combine the strengths of exact techniques such as Pseudo-Boolean (PB) solvers to solve the NP-complete problem of finding a feasible implementation with a meta-heuristic such as an Evolutionary Algorithm (EA) that performs the actual multi-objective DSE. The chapter also shows how to extend the PB solving in such a way that both linear and also nonlinear system constraints such as a maximum latency can be considered.

Besides the need for suitable exploration algorithms for DSE, the design space of the system under design becomes increasingly complex: It may not only arise from design degrees of freedom of one but from several different layers of abstraction. ▸ Chapter 8, "Architecture and Cross-Layer Design Space Exploration", introduces the challenges that arise when having to handle multiple layers of abstraction concurrently and presents ways to achieve a so-called *cross-layer* DSE. Performing simulations including several layers of abstraction to determine the quality numbers of an implementation may come at significant execution times. As a remedy, the chapter discusses techniques to prune the large design space and applies predictive models to avoid the mentioned expensive simulations.

Traditionally, DSE in the context of HW/SW codesign has targeted (embedded and/or safety-critical) systems with a known set of applications to be executed in a more or less recurring or *static* fashion. Recently, the rise of multi- and many-core architectures and the growing demand for energy efficiency require to not always overdesign systems for a static worst-case usage, but dynamic behavior at run-time

is explicitly allowed and often even appreciated. ▸ Chapter 9, "Scenario-Based Design Space Exploration", introduces the concept of *scenario-based design* where the dynamic behavior of applications is categorized into so-called *scenarios* which dynamically change at run-time. This chapter presents an approach for the DSE of such systems that tries to optimize a system for an average-case behavior across all different scenarios using a multi-objective Genetic Algorithm (GA). Since the combination of all different scenarios of all applications would result in a huge space, the chapter introduces a technique to predict the quality numbers of an implementation using a small subset of *representative* scenarios. While ▸ Chap. 9, "Scenario-Based Design Space Exploration" models the dynamic run-time behavior by means of scenarios at design time, ▸ Chap. 10, "Design Space Exploration and Run-Time Adaptation for Multicore Resource Management Under Performance and Power Constraints", goes one step further and introduces DSE and run-time adaptation techniques that are applied at run-time. As such, the problems of resource allocation, task binding, scheduling, etc., remain the same but have to be dynamically adapted by means of *resource management*. Resource management particularly takes place when quality numbers then formulated as power and performance constraints may be violated due to changing application mixes and/or workloads of the system. The chapter gives a detailed overview of different resource management strategies (centralized, decentralized, design time, run-time, etc.), discusses the advantages and disadvantages, and details the most important optimization goals and constraints. Moreover, it particularly discusses the trade-off between performance requirements and available power budgets.

Overall, DSE has become a key ingredient of successful HW/SW codesign. Often completely automatically, it reveals existing trade-offs regarding crucial quality numbers to the designer. This not only enhances the designer's insight into the design space of the system but also the trust in design decision to be made early in the development process.

1.4 Processor, Memory, and Communication Architecture Design

Hardware/Software Codesign involves exploiting complementary trade-offs of hardware and software. Thus hardware is an integral and essential piece of the puzzle in the codesign of electronic systems. High-performance computing is dominated by complex general-purpose processors as hardware and primary innovations happen in software. In contrast, HW/SW codesign enables both the software and the hardware to be customized and optimized for a specific application(s). The prescient knowledge about the applications on the target system opens up opportunity to venture beyond traditional general-purpose processors, memory, and interconnect. Part 4 "Processor, Memory, and Communication Architecture Design" of this handbook, *Processor, Memory, and Communication Architecture Design* provides a glimpse into the unique hardware resources available for HW/SW codesign.

The common theme here is hardware platforms that are amenable to application-specific design.

The processor engine in codesigned platforms has initially been either tiny microcontrollers and/or Application-Specific Integrated Circuits (ASICs) custom designed for a specific functionality. With the advancement of Moore's law, microcontrollers have been replaced by powerful general-purpose microprocessors that are far more cost-effective compared to ASIC accelerators. Software programmability of general-purpose processors enables the same architecture to be reused across a large class of applications. But they also lack the performance and energy efficiency of ASICs, which are essential for certain electronic systems. At the same time, the enormous nonrecurring engineering cost precludes the choice of ASICs in cost-sensitive designs. This gulf between the two extremes is bridged through reconfigurable computing and application-specific processors that allow the designer to achieve ASIC-like performance and energy efficiency with processor-like application development cost. These architectures are also excellent examples of hardware/software codesign principles being applied effectively. While the efficiency of the processor is an important aspect, the memory plays an equally influential role in system power and performance and should be considered in all levels of the system design. Again, the codesign paradigm brings about some nontraditional memory structures. The examples include reconfigurable and/or customized caches, software-controlled scratchpad memory, and application-specific register file designs. Similarly, emerging nonvolatile memory technologies made early appearance in codesigned systems because the memory hierarchy can be designed to custom-fit the application profile, thereby alleviating several challenges associated with exploiting nonvolatile memory in general-purpose systems. Finally, while the initial generation of electronic systems relied on only a single processor or ASIC, the complexity of contemporary systems demands multi-core or even many-core architectures to take advantage of the concurrency present in most applications and provide the required performance under the power constraints. Thus, the hardware platforms in codesigned systems today consist of a number of processors, reconfigurable logic, ASIC accelerators, and different memory structures that need to communicate with each other. The Network-on-Chip (NoC) provides a flexible communication substrate to take care of the diverse communication requirements. Similar to processor and memory, the codesign methodology allows application-specific NoC synthesis to better match the application demands.

In this handbook, ▸ Chap. 11, "Reconfigurable Architectures", presents an introduction to reconfigurable architecture that combines the flexibility of software with the high-performance and low-power advantages of hardware. The heart of any reconfigurable architecture is the reconfigurable fabric that can be reprogrammed after fabrication to implement and accelerate computational kernels with sufficient parallelism. This chapter presents two popular classes of reconfigurable architectures: Field-Programmable Gate Arrays (FPGAs) and Coarse-Grained Reconfigurable Arrays (CGRAs). ▸ Chapter 12, "Application-Specific Processors", follows up this domain-specific design theme with application-specific processors that parameterize

and augment the underline base architecture with application-specific features such as custom instructions, suitable register file and cache size, etc. Application-specific processors also attempt to bridge the gap between flexibility and performance. However, unlike reconfigurable architectures that require specialized programming environments, application-specific processors, once designed appropriately, can be utilized in familiar software programming environments.

On the memory side, ▸ Chap. 13, "Memory Architectures", delves into memory hierarchy in system-on-chip designs. The chapter discusses relative merits of different memory structures, and customization of the memory hierarchy to improve power-performance characteristics of an application is highlighted. Compiler optimizations while mapping application data into memory hierarchy are also presented. ▸ Chapter 14, "Emerging and Nonvolatile Memory", offers a sneak peek into the future with emerging and nonvolatile memories that have the ability to retain information even after the power has been switched off. This nonvolatility contributes to several advantages including lower leakage power and higher density. However, the relatively higher write cost and potential endurance issues have so far plagued widespread adoption of emerging memories. The chapter introduces different available emerging memory technologies and the optimizations to improve write latency and endurance through hybrid memory structures combining emerging memory technology with traditional CMOS memory.

Following processing and memory, ▸ Chap. 15, "Network-on-Chip Design", introduces Network-on-Chip (NoC) designs that enable communication among processors and memory on chip. The chapter describes various NoC architectures. Special emphasis is placed on power optimizations for NoC, optimized application mapping on NoC, as well as customization of NoC according to application-specific communication patterns. Finally, ▸ Chap. 16, "NOC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications", puts together the concepts presented in the previous chapters in designing a Multi-Processor System-on-Chip (MPSoC) architecture. This chapter shows how different hardware resources can be combined together and virtualized to provide a flexible hardware platform that can be shared across multiple applications in the context of mixed time-critical systems.

1.5 Hardware/Software Cosimulation and Prototyping

In contrast to the traditional design flow with separate HW and SW development, as discussed in Fig. 1.1, Hardware/Software Codesign relies on a *model* of the target system during the product development. This system model describes both the HW and SW parts of target system and may consist of hardware and software parts itself. Here, the building of a hardware model is typically referred to as *prototyping*, whereas the pure software modeling and execution is called virtual prototyping or simply *simulation*. The execution of the combined codesign system model with both HW and SW parts is correspondingly termed *cosimulation*.

Both HW/SW cosimulation and prototyping are essential steps in the modern codesign flow. As illustrated in Fig. 1.2 above, the main objectives of cosimulation

and prototyping are performance estimation and functional validation of the target model, so that the system designer can make educated design decisions for platform selection, HW/SW partitioning, and algorithm and architecture optimization.

In Part 5, "Hardware/Software Cosimulation and Prototyping", four dedicated chapters provide an in-depth description of several state-of-the-art approaches to modern cosimulation and prototyping.

▸ Chapter 17, "Parallel Simulation", presents classic and modern simulation techniques for codesign models described in System-Level Description Languages (SLDLs), such as the IEEE SystemC standard which is widely used in industry and academia. Emphasizing the topic of *Parallel Simulation*, the chapter first reviews the classic sequential Discrete Event Simulation (DES) approach and then focuses on advanced Parallel Discrete Event Simulation (PDES) algorithms. Parallel simulation approaches can run multiple simulation threads in parallel on multi- or many-core processor hosts and thus gain execution speed by an order of magnitude for system models that exhibit parallel structures. Observing PDES semantics accurately, however, is not easy. The chapter describes the complexities of analyzing inter-thread dependencies in detail and finally presents a state-of-the-art approach called *Out-of-Order Parallel Discrete Event Simulation (OOO PDES)* which can break the traditional simulation cycle barriers and execute threads in parallel and out of order (ahead of time) while maintaining the standard SystemC modeling semantics.

▸ Chapter 18, "Multiprocessor System-on-Chip Prototyping Using Dynamic Binary Translation" presents *Multi-Processor System-on-Chip Prototyping Using Dynamic Binary Translation*. Binary translation is a processor emulation technique that enables the efficient execution of a binary software program on a simulation host processor. Notably, the Instruction-Set Architecture (ISA) of the host processor can be different than the ISA of the target processor, so that, for example, a 64-bit workstation processor can execute a binary program compiled for a 16-bit microcontroller. The chapter first provides a brief overview of dynamic binary translation techniques and the peculiarities involved, such as the advanced support for Single Instruction, Multiple Data (SIMD) instructions and Very Long Instruction Word (VLIW) architectures. Next, it also covers improvements to the translation process in order to monitor nonfunctional metrics for performance estimation and finally describes the seamless integration with virtual prototyping platforms.

▸ Chapter 19, "Host-Compiled Simulation" introduces *Host-Compiled Simulation* which is based on Virtual Prototypes (VPs). Such abstract software platform models enable early software development, validation, and exploration before the availability of any actual target hardware platform. The chapter covers several new approaches that overcome the accuracy-speed bottleneck of today's virtual prototyping methods. Such next-generation VPs utilize sophisticated host-compiled software models which can achieve both high speed and high accuracy in the timing behavior of the application, operating system, and underlying hardware architecture. Special attention is paid to improved simulation speed by using so-called TLM-based communication models.

After the general host-compiled simulation discussion, ▶ Chap. 20, "Precise Software Timing Simulation Considering Execution Contexts" covers an important optimization technique focusing on the critical timing accuracy of high-level models. The chapter describes a novel approach for *Precise Software Timing Simulation Considering Execution Contexts*. Classic timing estimation in simulation models is agnostic to the exact execution context of the software running on the modeled processors. In contrast, context-sensitive simulation enables a precise approximation of software timing while maintaining high simulation speed. The chapter provides an overview of this concept and presents a state-of-the-art context-sensitive simulation framework. For this novel approach, the experimental results show accurate and fast timing simulation for software executing on current commercial embedded processors with complex high-performance microarchitectures.

1.6 Performance Estimation, Analysis, and Verification

As illustrated in Fig. 1.2, any framework aimed at assisting developers in building efficient and high-confidence systems has to provide appropriate support in order to check that the resulting HW/SW implementation works correctly. This notion of correctness has to address both *functional* aspects of the system and *nonfunctional* ones, such as *timing properties*.

The first step toward meaningful system verification is a support for specifying the requirements (properties) of the designed system. With other words, we need to unambiguously describe what are the desired/undesired behaviors of the HW/SW implementation. Once such a *system specification* is at hand, the next step is to check that it is correctly implemented by the existing design. Traditionally, system specifications were expressed in natural language which, due to unavoidable ambiguities, cannot be used as an entry for rigorous verification. What is needed is a specification based on a mathematically rigorous notation for expressing system requirements. One such formalism, which has gained wide popularity both in the hardware and software community, is *temporal logic* [11].

The challenging problem now is to check if the system satisfies its temporal logic specification. One way to perform this checking, in particular in the case of safety-critical systems, is by *formal verification*. This can be done, for example, by using *model checking* techniques [12]. While such formal verification techniques have been successfully used in practice, in many cases, their application can become difficult due to tractability problems. Therefore, in particular in the case of noncritical applications, verification is based on *testing* the system using simulation and running the system implementation. When applied in a systematic way and using appropriate coverage metrics, such techniques have been very successful in practice. Moreover, formal techniques can be used as part of such testing frameworks, for specification of the properties to be verified, and in order to systematically generate efficient test inputs.

Initially, the target of techniques like the ones discussed above has been functional verification. However, with the proliferation of embedded and cyber-physical

systems, in which timeliness has become an intrinsic part of the notion of correctness, verifying timing properties has gained more and more attention. Several problems have to be addressed in this context. One is to determine the execution time of a piece of software (task) when running on a certain processor. One obvious technique is to measure the execution time by running the task. However, execution time depends, among others, on the inputs applied during measurement. Therefore, if the Worst-Case Execution Time (WCET) is of interest, a measurement-based technique might not be safe. Hence, formal techniques to determine WCETs of tasks have been proposed [10,33]. They are based on static analysis of the code and on models of the processor microarchitecture. The WCET, as determined above, reflects the behavior of the task when running alone on the system, without sharing any resources. In modern systems, however, a potentially large number of tasks are running together, sharing not only processor but also other resources such as memory and communication infrastructure. Such an interference between tasks makes the analysis of timing properties a challenging problem. Moreover, modern execution platforms consist of distributed processors interconnected by complex networks, which makes the analysis even more complex. In this context, formal techniques using model checking have been extended to support the verification of timing properties for systems specified as *timed automata* [2,6]. Another family of formal techniques is based on *scheduling analysis* of real-time applications specified as task sets [8]. One of the main challenges facing current research in this area is to capture the complex resource sharing patterns characteristic to modern multi-core platforms.

In this handbook, ▸ Chap. 22, "Semiformal Assertion-Based Verification of Hardware/Software Systems in a Model-Driven Design Framework", addresses the issue of functional verification of HW/SW systems using simulation-based techniques combined with formal specification. The approach is based on (1) a *model-driven design* technique with automata-based modeling, (2) formal *assertions* specifying desired properties, and (3) automatic test stimuli generation and mutant-based quality evaluation. The assertion definitions are automatically synthesized into executable checkers that are integrated into the simulation environment and monitor the software execution for detecting violations of the imposed requirements.

▸ Chapters 21, "Timing Models for Fast Embedded Software Performance Analysis", ▸ 23, "CPA: Compositional Performance Analysis", and ▸ 24, "Networked Real-Time Embedded Systems" are dealing with the verification of timing properties. ▸ Chapter 21, "Timing Models for Fast Embedded Software Performance Analysis", presents a simulation-based approach for determining the temporal behavior of software modules. The technique can be efficiently used, particularly early in the design process, in order to make initial system configuration decisions. The basic idea is to build timing models that are integrated with the functional simulation environment. Different mechanisms are presented for generating those timing models, depending on the degree to which the details of the underlying hardware platform are known.

While the approach presented in ▸ Chap. 21, "Timing Models for Fast Embedded Software Performance Analysis" is meant for early performance estimation,

▸ Chaps. 23, "CPA: Compositional Performance Analysis" and ▸ 24, "Networked Real-Time Embedded Systems" are addressing issues related to the formal *response time analysis* for safety-critical hard real-time systems. ▸ Chapter 23, "CPA: Compositional Performance Analysis", introduces a *compositional timing analysis* approach that supports the analysis of complex systems consisting of various modules using different scheduling and arbitration strategies. Due to its scalability and the availability of industrial strength tools, the framework is widely used in practice, especially for the analysis of distributed applications in automotive systems. And this brings us to ▸ Chap. 24, "Networked Real-Time Embedded Systems", which is dedicated to the topic of networked real-time embedded systems. Distributed embedded systems are becoming increasingly common, and their deign has to consider not only the HW/SW aspects but also their integration with the underlying *communication network*. The chapter provides an introduction to some of the real-time communication networks dominant in today's automotive industry or predicted to be widely used in the future: *CAN*, *FlexRay*, and *Switched Ethernet*. The basic features of these protocols are presented as well as some of the related formal timing analysis techniques.

1.7 Hardware/Software Compilation and Synthesis

After design space exploration and architecture fine-tuning, we need to finalize the implementation of software and hardware, by performing one last step – hardware and software compilation and synthesis. In system design, synthesis means constructing an implementation of software or hardware that provably satisfies a given high-level specification. To be more specific, software synthesis, or compilation, transforms source code written in a programming language (such as C/C++ or JAVA) into a binary form that can be recognized by target machines [1]. Similarly, hardware synthesis interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior [13]. The goal of this step is therefore to translate the specifications of hardware and software from the previous steps of the hardware/software codesign process into a real implementation of the hardware platform and the machine instructions that run efficiently on the hardware, respectively.

The compilation of software should take into consideration the hardware parameters, since one very important advantage of HW/SW codesign is that it allows software requirements decide choices of hardware parameters, and hardware design parameters in turn motivate changes in the software design to better make use of these hardware parameters. If the compilation of software is agnostic to the underlying hardware details, then such codesign efforts become vain. Unfortunately, the popular optimizing compilers used today, such as GCC [30] or LLVM-based compilers [24], are not aware of many hardware details. Being oblivious to hardware details is fine for most general-purpose processors, but it can result in leaving a lot of power and performance optimization opportunities on the table for resource-constrained embedded systems. Consider a processor without hardware branch

predication to preserve power efficiency [16]. To reduce the performance loss caused by hardware branch prediction, such processors usually provide instructions for software branch hinting. If a compiler ignores such hardware feature and does not insert any software branch hints, then the penalties due to branch instructions can result in a huge performance loss. Another example is that some processors may have scratchpad memories (SPMs) [5, 28] instead of or along caches in the memory hierarchy. Missing out such a feature of the memory subsystem could cost a considerable amount of performance, as exploiting the SPMs could largely improve the system performance. For example, placing frequently executed instructions or data in the SPMs instead of leaving them to caching may avoid unnecessary cache misses due to cache pollution. Fortunately, there have been many research efforts toward hardware-aware and memory-aware compilation. A large body of memory management of SPM-based architectures is a prominent example. The earliest SPM management techniques were mostly static, assigning the most frequently accessed part of application data into an SPM and not changing the location of data once it is copied into the SPM. The static approaches do not take into consideration the dynamic behaviors of programs and are gradually replaced by dynamic SPM management techniques over time. Dynamic SPM management techniques copy data between an SPM and the main memory dynamically at run-time and the location of data in SPM may be changed as it is swapped in and out from SPM. The goals of these SPM management techniques vary from worst-case execution time, average-case execution time, and power efficiency.

In this handbook, ▸ Chaps. 25, "Hardware-Aware Compilation", and ▸ 26, "Memory-Aware Optimization of Embedded Software for Multiple Objectives", introduce compiler designs that are aware of underlying microarchitectural features and memory hierarchy, with details of some of the techniques, respectively. The results show that performing optimization with regard to hardware characteristics can significantly improve design objective, be it improving performance or reducing worst-case execution time.

On the other hand, the synthesis of hardware should implement the hardware in a way that can meet the design constraints posed by the target (software) application. These common constraints include area, power, temperature, performance, and reliability. It is readily seen that a high-quality hardware platform should function properly following all these constraints. Otherwise, the outcome will not be satisfying – even if a mobile device can deliver impressive performance running applications, it may still be undesirable to consumers if running any application drains battery power abnormally fast, for example. The implementation of a high-quality hardware platform is, however, complicated by conflicts of interests among these constraints. For example, increasing performance (flops) of a processor may end up increasing cost of area and power consumption. Therefore, implementation of the hardware cannot simply optimize toward one design objective. Instead, the relation and trade-offs of different constraints should be considered at the same time in order to obtain a (nearly) optimal hardware implementation that satisfies all the constraints. To illustrate how to implement such a hardware platform, ▸ Chap. 27, "Microarchitecture-Level SoC Design", elaborates how to construct a

System-on-Chip (SoC), meeting constraints of performance, power, thermal, and reliability. SoC needs a system design methodology that combines predesigned and preverified components, called Intellectual Property (IP) cores into a single chip [29]. An IP core can be an embedded processor, a memory block, or any component that handles application-specific processing functions. By reusing IP cores, SoCs can effectively overcome the increased complexity of chips.

1.8 Codesign Tools and Environments

In previous sections of this chapter and in the main parts of this book, various ingredients for successful HW/SW codesign methodologies as originally summarized in Fig. 1.2 are described. Over the years, many of the underlying methods and concepts have been realized in practical academic or industrial codesign tools. Most of the early tools were thereby focused on providing point solutions for individual design tasks. An extensive survey of such point tools can be found in [14]. Only in later years, however, did comprehensive design environments aimed at supporting a complete HW/SW codesign flow start to appear [18]. Nevertheless, providing a comprehensive design automation solution that truly covers all aspects of a design flow aimed at automatically synthesizing an abstract, high-level system specification into a system implementation spanning across hardware, and software boundaries is a daunting, complex, and challenging endeavor. As such, existing codesign environments each have their unique focus, strengths, and weaknesses.

Chapters in Part 8, "Codesign Tools and Environment", describe several prominent examples of such codesign environments covering the spectrum of solutions as developed in both academia and industry. Tools differ in the supported specification models/languages, target architectures, and the amount of automation provided for different design tasks. However, there are also many common characteristics that have emerged and are shared by existing tools, such as the use of generally task-based concurrency models as input descriptions and the support for heterogeneous, bus-based MPSoC architectures as targets for synthesis.

▶ Chapters 28, "MAPS: A Software Development Environment for Embedded Multicore Applications", ▶ 29, "HOPES: Programming Platform Approach for Embedded Systems Design", ▶ 30, "DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips", and ▶ 31, "SCE: System-on-Chip Environment" describe four codesign environments that originated in academic settings. ▶ Chapter 28, "MAPS: A Software Development Environment for Embedded Multicore Applications" presents the *MPSoC Application Programming Studio (MAPS)*. The focus in MAPS is on software aspects, with the goal of providing a comprehensive environment to develop, debug, and deploy software for heterogeneous Multi-Processor System-on-Chip (MPSoC) architectures containing multiple concurrent processors of differing type. Software development for such concurrent, heterogeneous architectures is a challenge that MAPS aims to address. MAPS is based on a variant of Process Networks (PNs) as input programming model, with support for automatic code parallelization, early performance

estimation, and final heterogeneous target code generation. MAPS technology has also been spun out from its original academic origins into a startup company.

▶ Chapter 29, "HOPES: Programming Platform Approach for Embedded Systems Design" describes the *Hope of Parallel Software (HOPES)* environment. HOPES builds on and is the successor to the *Ptolemy extension as a Codesign Environment (PeaCE)* previously developed in the same group. PeaCE and HOPES use variants, combinations, and extensions of Synchronous Data Flow (SDF) and Finite-State Machine (FSM) computational models as input descriptions. While PeaCE was focused on classical HW/SW codesign, partitioning, and interfacing tasks targeting a single CPU assisted by a set of hardware accelerators, HOPES extends on this framework to target more advanced multiprocessor mapping, scheduling, and code generation tasks. This is similar to the goals in MAPS, with a particular emphasis on support for design space exploration.

▶ Chapter 30, "DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips" introduces the *Daedalus* environment. In Daedalus, applications are specified in the form of extended Kahn Process Network (KPN) models, where automatic parallelization from sequential input code is supported similar to MAPS. Daedalus then employs a library-based approach to synthesize a target MPSoC that is assembled out of predesigned heterogeneous processor, memory, hardware Intellectual Property (IP), and communication/bus components, including generation of all required glue logic and target processor code. In the process, Daedalus performs optimized application mapping and target architecture synthesis using an automated design space exploration approach that is aided by cosimulation and performance estimation tools.

Finally, ▶ Chap. 31, "SCE: System-on-Chip Environment", provides an overview of the *System-on-Chip Environment (SCE)* as the final academic tool set described in Part 8, "Codesign Tools and Environment". SCE is based on the *SpecC* SLDL, which is used both as input specification as well as internal representation of all intermediate design models within the tool itself. In SCE, all decision-making and design space exploration for application mapping and target architecture definition is manual. By contrast, the emphasis is on automated computation and communication synthesis for complex and general MPSoC target architectures, including synthesis of optimized software, Operating System (OS) and driver stacks, custom hardware components, hardware interfaces, and bus architectures. Using internally a successive and gradual refinement process on top of its SpecC basis, SCE also supports automatic generation of cosimulation and performance models at varying levels of abstraction. A derivative of SCE called the *Specify-Explore-Refine (SER)* environment has been transferred into an industry setting through collaborations with the Japanese Aerospace Exploration Agency (JAXA).

▶ Chapters 32, "Metamodeling and Code Generation in the Hardware/Software Interface Domain", ▶ 33, "Hardware/Software Codesign Across Many Cadence Technologies", and ▶ 34, "Synopsys Virtual Prototyping for Software Development and Early Architecture Analysis", then describe three codesign approaches used or originating in industry. ▶ Chapter 32, "Metamodeling and Code Generation in the Hardware/Software Interface Domain", provides an industrial perspective

on the practical use of codesign methodologies and concepts in a semiconductor company. The chapter describes a *metamodeling* approached used in Infineon for MPSoC architecture definition and code generation. Using languages like IP-XACT or UML/SysML, a metamodel defines an architecture in abstract, annotated, and extended netlist form from which concrete models in the form of SystemC, RTL, C, or other code can be automatically generated. Using a single architecture specification, this allows otherwise incompatible tools to be used and combined for further system implementation and synthesis across hardware and software boundaries.

▶ Chapters 33, "Hardware/Software Codesign Across Many Cadence Technologies", and ▶ 34, "Synopsys Virtual Prototyping for Software Development and Early Architecture Analysis", finally provide the perspectives of two major Electronic Design Automation (EDA) tool vendors on codesign solutions offered by their companies, namely, Cadence and Synopsys, respectively. This includes various point tools for cosimulation, performance and power modeling, verification, FPGA-based emulation, high-level hardware synthesis and application-specific processor design, as well as complete solutions for integrated system and hardware and software development as supported by a common virtual prototyping basis on top of industry-standard languages like SystemC.

1.9 Applications and Case Studies

Early work on specialized codesign techniques included a focus on the areas of control-dominated systems and digital signal processing (DSP). For example, Antoniazzi et al. introduced a codesign toolset targeted to control-oriented ASICs, such as those applied in digital switching systems for telecommunications [3]. The toolset operated on system models that were specified in terms of hierarchical, concurrent finite-state machines. The tool emphasized trade-off exploration involving transformations for restructuring application processes, clustering of processes onto architectural units, and binding of these architectural units into hardware or software implementations. The toolset was integrated with a commercial design environment called speedCHART, which was based on the Statecharts formalism [20].

Kalavade and Lee presented a codesign methodology for signal processing and communication systems. The methodology applied the Ptolemy tool for heterogeneous modeling and design [23] and SDF model of computation. The authors' approach focused on enabling interoperability and integrated design space exploration using advanced tools that are effective for critical aspects of the signal processing codesign process, including multiprocessor software synthesis, custom hardware synthesis, digital hardware modeling and simulation, and SDF modeling of analog components, such as A/D and D/A converters.

Part 9, "Applications and Case Studies" of this handbook, *Applications and Case Studies*, focuses on the demonstration of codesign methodologies and tools in the context of specific applications areas and concrete case studies. It contains six chapters that provide reviews of state-of-the-art methodologies, tools, platforms,

and implementations for codesign in the context of important applications and application domains. A common theme throughout the chapters is the representation and exploitation of relevant characteristics in the targeted applications to streamline codesign processes and bridge general-purpose codesign methods with constraints and design objectives that are specific to the applications.

▶ Chapter 35, "Joint Computing and Electric Systems Optimization for Green Datacenters", presents a real-time framework for jointly minimizing the energy consumption and carbon footprint of green virtualized data centers. The framework involves two cooperating subsystems: the Datacenter Energy Controller, which minimizes energy consumption while preserving quality of service, and the Green Energy Controller, which manages renewable energy sources and electrical energy storage systems that incorporate heterogeneous battery technologies.

▶ Chapter 36, "The DSPCAD Framework for Modeling and Synthesis of Signal Processing Systems" presents a computer-aided design framework, called the DSPCAD Framework, that facilitates design, implementation, and optimization of signal processing systems using data-flow models of computation. The framework includes three complementary tools – the Data-flow Interchange Format (DIF) for data-flow modeling, the LIghtweight Data-flow Environment (LIDE) for actor implementation, and the DSPCAD Integrative Command Line Environment (DICE) for cross-platform DSP system integration and validation.

▶ Chapter 37, "Control/Architecture Codesign for Cyber-Physical Systems", discusses the emerging paradigm of control/architecture codesign, which involves the joint design of control parameters and embedded platform parameters for cyber-physical systems. Three examples of control/architecture codesign approaches are presented together with case studies demonstrating the approaches. These approaches involve, respectively, communication-aware, memory-aware, and computation-aware design.

▶ Chapter 38, "Wireless Sensor Networks", discusses challenges in design flows for Wireless Sensor Networks (WSNs) and reviews various development techniques and tools for WSNs. The authors also present in detail two specific model-based design flows with varying degrees of automation and their associated toolsets. The flows are demonstrated through two case studies involving, respectively, a self-powered WSN gateway and a WSN sensor node for air quality monitoring.

▶ Chapter 39, "Codesign Case Study on Transport-Triggered Architectures", presents a codesign case study involving the implementation of the Fast Fourier Transform (FFT) using customizable architecture templates that are defined in terms of the Transport Triggered Architecture (TTA) paradigm. In the TTA paradigm, programs are executed entirely in terms of data transport (move instructions), and operations are carried out as side effects of these transports. The case study presented in this chapter applies a toolset called the TTA-based Codesign Environment (TCE), which supports optimized design of customized TTA architectures from high-level programming languages.

▶ Chapter 40, "Embedded Computer Vision", finally reviews design methodologies, platform architectures, and application-specific architectures that are relevant

to codesign for embedded computer vision. The chapter motivates the utility of heterogeneous architectures and multiprocessor systems on chip for this application domain and discusses application-specific architectural solutions for a variety of important embedded computer vision applications, including foreground detection, face detection and recognition, and convolutional neural networks.

1.10 Conclusion

In this introductory chapter, an overview of the remaining contents in the handbook is presented. The various chapters of this book are organized into parts that align with the discussions in the different subsections presented earlier in this chapter. We hope that you enjoy reading the book as much as we enjoyed preparing it.

References

1. Aho AV, Lam MS, Sethi R, Ullman JD (2006) Compilers: principles, techniques, and tools, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston
2. Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235
3. Antoniazzi S, Balboni A, Fornaciari W, Sciuto D (1994) A methodology for control-dominated systems codesign. In: Proceedings of the international workshop on hardware/software codesign, pp 2–9
4. Balarin F, Chiodo M, Giusto P, Hsieh H, Jurecska A, Lavagno L, Passerone C, Sangiovanni-Vincentelli A, Sentovich E, Suzuki K, Tabbara B (1997) Hardware-Software co-design of embedded systems: the POLIS approach. Kluwer Academic Publishers, Boston
5. Banakar R, Steinke S, Lee BS, Balakrishnan M, Marwedel P (2002) Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In: Proceedings of CODES
6. Bengtsson J, Larsen K, Larsson F, Pettersson P, Yi W (1996) UPPAAL—a tool suite for automatic verification of real-time systems. Springer, Berlin, pp 232–243. doi:10.1007/BFb0020949
7. Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages twelve years later. Proc IEEE 91(1):64–83
8. Buttazzo GC (2011) Hard real-time computing systems: predictable scheduling algorithms and applications, vol 24. Springer, New York
9. Cassandras C, Lafortune S (2008) Introduction to discrete event systems, 2nd edn. Springer, New York
10. Chattopadhyay S, Roychoudhury A, Rosén J, Eles P, Peng Z (2014) Time-predictable embedded software on multi-core platforms: analysis and optimization. Found Trends®Electron Des Autom 8(3–4):199–356. doi:10.1561/1000000037
11. Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8(2):244–263
12. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge
13. Coussy P, Morawiec A (2008) High-level synthesis: from algorithm to digital circuit, 1st edn. Springer, Dordrecht
14. Densmore D, Passerone R, Sangiovanni-Vincentelli A (2006) A platform-based taxonomy for ESL design. IEEE Des Test Comput 23:359–374
15. Edwards S, Lavagno L, Lee EA, Sangiovanni-Vincentelli A (1997) Design of embedded systems: formal models, validation, and synthesis. Proc IEEE 85(3):366–390

16. Eichenberger AE, O'Brien JK, O'Brien KM, Wu P, Chen T, Oden PH, Prener DA, Shepherd JC, So B, Sura Z, Wang A, Zhang T, Zhao P, Gschwind MK, Archambault R, Gao Y, Koo R (2006) Using advanced compiler technology to exploit the performance of the cell broadband engine™ architecture. IBM Syst J 45:59–84
17. Gajski DD, Dutt N, Wu A, Lin S (1992) High level synthesis: introduction to chip and system design. Springer, New York
18. Gerstlauer A, Haubelt C, Pimentel A, Stefanov T, Gajski D, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput Aided Des Integr Circuits Syst 28(10):1517–1530
19. Girault A, Lee B, Lee EA (1999) Hierarchical finite state machines with multiple concurrency models. IEEE Trans Comput Aided Des Integr Circuits Syst 18(6):742–760
20. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274
21. Hoare C (1985) Communicating sequential processes. Prentice Hall, Englewood Cliffs
22. Kahn G (1974) The semantics of a simple language for parallel programming. In: Proceedings of IFIP congress 74. North-Holland Publishing Co
23. Kalavade A, Lee, EA (1993) A hardware/software codesign methodology for DSP applications. IEEE Des Test Comput 10(3):16–28
24. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 international symposium on code generation and optimization (CGO'04), Palo Alto
25. Lee EA, Parks TM (1995) Dataflow process networks. Proc IEEE 83:773–799
26. Lee EA, Sangiovanni-Vincentelli A (1998) A framework for comparing models of computation. IEEE Trans Comput Aided Des Integr Circuits Syst 17(12):1217–1229
27. Prakash S, Parker AC (1992) SOS: synthesis of application-specific heterogeneous multiprocessor systems. J Parallel Distrib Comput 16(4):338–351
28. Redd B, Kellis S, Gaskin N, Brown R (2014) The impact of process scaling on scratchpad memory energy savings. J Low Power Electron Appl 4(3):231. http://www.mdpi.com/2079-9268/4/3/231
29. Saleh R, Wilton S, Mirabbasi S, Hu A, Greenstreet M, Lemieux G, Pande PP, Grecu C, Ivanov A (2006) System-on-chip: reuse and integration. Proc IEEE 94(6):1050–1069. doi:10.1109/JPROC.2006.873611
30. Stallman RM, DeveloperCommunity G (2009) Using the GNU compiler collection: a GNU manual for GCC version 4.3.3. CreateSpace, Paramount
31. Teich J (2012) Hardware/Software codesign: the past, the present, and predicting the future. Proc IEEE 100(Special Centennial Issue):1411–1430. doi:10.1109/JPROC.2011.2182009
32. Teich J, Blickle T, Thiele L (1997) An evolutionary approach to system-level synthesis. In: Proceedings of the international workshop on hardware/software codesign (CODES/CASHE), pp 167–171
33. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenstrom P (2008) The worst-case execution time problem–overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3):Art. 36

Part II
Models and Languages for Codesign

Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems

2

Klaus Schneider and Jens Brandt

Abstract

Since the synchronous model of computation is shared between synchronous languages and synchronous hardware circuits, synchronous languages lend themselves well for hardware/software codesign in the sense that from the same synchronous program both hardware and software can be generated. In this chapter, we informally describe the syntax and semantics of the imperative synchronous language Quartz and explain how these programs are first analyzed and then compiled to hardware and software: To this end, the programs are translated to synchronous guarded actions whose causality has to be ensured as a major consistency analysis of the compiler. We then explain the synthesis of hardware circuits and sequential programs from synchronous guarded actions and briefly look at extensions of the Quartz language in the conclusions.

Acronyms

AIF	Averest Intermediate Format
EFSM	Extended Finite-State Machine
MoC	Model of Computation
SMV	Symbolic Model Verifier
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

K. Schneider (✉)
Embedded Systems Group, University of Kaiserslautern, Kaiserslautern, Germany
e-mail: schneider@cs.uni-kl.de

J. Brandt
Faculty of Electrical Engineering and Computer Science, Hochschule Niederrhein, Krefeld, Germany
e-mail: jens.brandt@hsnr.de

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_3

Contents

2.1 Introduction

Compared to traditional software development, the design of embedded systems is even more challenging: In addition to the correct implementation of the functional behavior, one has to consider also non-functional constraints such as real-time behavior, reliability, and energy consumption. For this reason, embedded systems are often built with specialized, often application-specific hardware platforms. To allow late design changes even on the hardware/software partitioning, languages and model-based design tools are required that can generate both hardware and software from the same realization-independent model. Moreover, many embedded systems are used in safety-critical applications where errors can lead to severe damages up to the loss of human lives. For this reason, formal verification is applied in many design flows using different kinds of formal verification methods.

The *synchronous Model of Computation (MoC)* [2] has shown to be well-suited to provide realization-independent models for a model-based design of embedded reactive systems where both hardware and software can be generated. There are at least the following reasons for this success: (1) It is possible to determine tight bounds on the reaction time by a simplified worst-case execution time analysis [28,29,31,32,50], since by construction of the programs, only a statically bounded finite number of actions can be executed within each reaction step. (2) The formal semantics of these languages allows one to prove (a) the correctness of the compilation and (b) the correctness of particular programs with respect to given formal specifications [40,42,44,47]. (3) It is possible to generate both efficient software and hardware from the same synchronous programs. Since the synchronous MoC is also used by synchronous hardware circuits, the translation to synchronous hardware circuits [3,4,38–40,47] is conceptually clearer and simpler than for classic hardware description languages such as VHSIC Hardware Description Language (VHDL) or Verilog (at least if these are not restricted to synchronous synthesizable subsets).

All these advantages are due to the *synchronous MoC* that postulates two essential properties: (1) A run of a synchronous system consists of a linear sequence of discrete reactions. In each of these reactions, a synchronous program reads the inputs, updates the internal state, and computes the values of the corresponding outputs. (2) All the computations within such a reaction are virtually performed in zero time, i.e., they are supposed to happen at the very same point of time so that every computation can immediately see the effects of every other computation within the same reaction step. Of course, this is not possible in a real system, but executing all computations according to the underlying data dependencies gives the programmer the impression postulated by the synchronous MoC.

The synchronous MoC simplifies the design of reactive embedded systems, since developers do not have to care about low-level details like timing, synchronization, and scheduling. Instead, the synchronous paradigm poses some specific problems to compilers, in particular, the *causality analysis* [4, 7, 13, 24, 34, 45, 48, 52] of synchronous programs. Intuitively, causality cycles occur when the input of a computation depends on its own output. Causally correct programs therefore have a causal order of the actions in every macro step, which allows the program to compute values before they are read. Another important analysis is the *clock consistency* that is required for synchronous programs with more than one clock: Here, we have to ensure that variables are only read at points of time when they are defined and that they are only assigned values whenever their clocks allow this. Although these and other problems turned out to be quite challenging, research over the last three decades has considered these problems in detail, found practical algorithms, and developed various compilers like [6, 11, 16, 17, 19, 21, 36, 37, 47] that are able to generate both hardware and software from one and the same synchronous program.

This chapter gives an overview of our synchronous language Quartz, which was derived from the pioneering language Esterel with a special focus on hardware design and formal verification. We start with a presentation of the language statements in Sect. 2.2 and explain with a few illustrative examples how it adheres with the synchronous MoC. The subsequent sections give more details of our Quartz compiler as implemented in our Averest system (http://www.averest.org): Sect. 2.3 shows how the compiler translates the source code into an intermediate representation which is subsequently used as the starting point for analysis and synthesis. Section 2.4 explains how causality is analyzed, before Sect. 2.5 describes how the intermediate representation is finally transformed to executable software or hardware. Finally, we briefly have a look at planned extensions of the language in the conclusions in Sect. 2.6.

2.2 The Synchronous Language Quartz

As outlined in the introduction, the synchronous MoC assumes that the execution consists of a sequence of reactions $\mathscr{R} = \langle R_0, R_1, \ldots \rangle$. In each reaction (also called macro step [25]), all the actions (also called micro steps) that take place

within a reaction are executed according to their data dependencies. This leads to the programmer's view that the execution of micro steps does not take time and that every macro step of a synchronous program requires the same amount of logical time. As a consequence, concurrent threads run in lockstep and automatically synchronize at the end of their macro steps, which yields the very important fact that even concurrent synchronous programs still have deterministic behaviors (indeed, most concurrent programming models lead to nondeterministic behaviors). As a result, *deterministic* single-threaded code can be obtained from multi-threaded synchronous programs. Thus, software generated from synchronous programs can be executed on ordinary microcontrollers without having the need of complex process scheduling of operating systems.

Synchronous hardware circuits are well-known models that are also based on the synchronous MoC: Here, each reaction is initiated by a clock signal, and all parts of the circuits are activated simultaneously. Although the signals need time to pass gates (computation) and wires (communication), propagation delays can be safely neglected as long as signals stabilize before the next clock tick arrives. Variables are either mapped to wires or registers, which both have unique values for every cycle. All these correspondences make the modeling and synthesis of synchronous circuits from synchronous programs very appealing [3, 4, 38–40, 47]. It is therefore not surprising that causality analysis of synchronous programs is a descendant of ternary simulation of asynchronous circuits [13, 33, 34, 52] and can this way be viewed as the proof of the abstraction from physical time to the abstract logical time (clocks).

In this section, we introduce our synchronous language Quartz [43], which provides the synchronous principle described above in the form of an imperative programming language similar to its forerunner Esterel [5, 6]. In the following, we give a brief overview of the core of the language that is sufficient to define most other statements as simple syntactic sugar. For the sake of simplicity, we do not give a formal definition of the semantics; the interested reader is referred to [43], which also provides a complete structural operational semantics in full detail. The Quartz core consists of the statements listed in Fig. 2.1, provided that S, S_1, and S_2 are also

`nothing`	(empty statement)
ℓ : `pause`	(start/end of macro step)
$x = \tau$ and `next`$(x) = \tau$	(assignments)
`if`(σ) S_1 `else` S_2	(conditional)
$S_1; S_2$	(sequence)
`do` S `while`(σ)	(iteration)
$S_1 \parallel S_2$	(synchronous concurrency)
`[weak] [immediate] abort` S `when`(σ)	(preemption: abortion)
`[weak] [immediate] suspend` S `when`(σ)	(preemption: suspension)
$\{\alpha\ x;\ S\}$	(local variable x of type α)
$inst$: $name(\tau_1, \ldots, \tau_n)$	(call of module *name*)

Fig. 2.1 The Quartz core statements

core statements, ℓ is a label, x and τ are a variable and an expression of the same type, σ is a Boolean expression, and α is a type.

Values of variables (or often called signals in the context of synchronous languages) of the synchronous program can be modified by assignments. They immediately evaluate the right-hand side expression τ in the current environment/macro step. Immediate assignments $x = \tau$ instantaneously transfer the obtained value of τ to the left-hand side x in the current macro step, whereas delayed assignments $\texttt{next}(x) = \tau$ transfer this value only in the following macro step. For this reason, all micro step actions are evaluated in the same variable environment which is also determined by these actions. The causality analysis makes sure that this cyclic dependency can be constructively resolved in a deterministic way.

The synchronous programming paradigm is therefore different to traditional sequential programs: For example, the incrementation of a loop variable i by an assignment $i = i + 1$ does not make sense in synchronous languages, since this requires to compute a solution to the (unsolvable) equation $i = i + 1$. Using delayed actions, one can write $\mathbf{next}(i) = i + 1$, which is the true intention of the assignment. Even more difficult is the interaction of several micro step actions, e.g., in the same sequence or in different parallel substatements: Their order given in the program does not matter for the execution, since execution just follows the data dependencies in a read-after-write schedule: In legal schedules, variables are only read if their value for the current macro step has already been determined. For example, the program in Fig. 2.2a has the same behavior as the program in Fig. 2.2b. Thus, every statement knows and depends on the results of all operations in the current macro step. In particular, a Quartz statement may influence its own activation condition (see the program in Fig. 2.2c). Obviously, this generally leads to causal cycles that have to be analyzed by the compilers, i.e., the compilers have to ensure that for all inputs, there is an execution order of the micro step actions such that a variable is never read before it is written in the macro step.

If a variable's value is not determined by an action in the current macro step, its value is determined by the so-called *reaction to absence*, which depends on the *storage type* of the variable. The current version of Quartz knows two of them: *memorized variables* keep the value of the previous step, while *event* variables are reset to a default value if no action determines their values. Future versions of Quartz will contain further storage types for hybrid systems and multi-clocked synchronous systems.

a	**b**	**c**
$a = 1;$	$b = a;$	$a = 1;$
$b = a;$	$a = 1;$	$\texttt{if}(b = 1)\ b = a;$
`pause;`	`pause;`	`pause;`
$a = b;$	$a = b;$	$\texttt{if}(a \neq 2)\ a = b;$

Fig. 2.2 Three Quartz programs illustrating the synchronous MoC

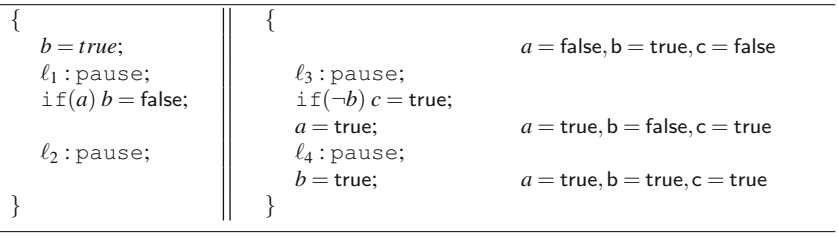


Fig. 2.3 Synchronous concurrency in Quartz

In addition to the usual control flow statements known from typical imperative languages (conditionals, sequences, and iterations), Quartz also offers synchronous concurrency. The *parallel statement* $S_1 \| S_2$ immediately starts the statements S_1 and S_2. Then, both S_1 and S_2 run in lockstep, i.e., they automatically synchronize when they reach their next `pause` statements. The parallel statement runs as long as at least one of the substatements is active.

Figure 2.3 shows a simple example consisting of two parallel threads with Boolean memorized variables a, b, and c. The threads are shown on the left-hand side of Fig. 2.3, and their effects are shown on the right-hand side. Initially, the default values of the variables are false, but in the first step, these values can be changed by the program. If the program is started, both threads are started. The first thread executes the assignment to b and stops at location ℓ_1, while the second thread immediately stops at location ℓ_3. In the second macro step, the program resumes from the labels ℓ_1 and ℓ_3. Thereby, the first thread cannot yet proceed since the value of a in this step is not yet known. The second thread cannot execute its if-statement either, but it can execute the following assignment to a. Thus, the second thread assigns true to a, then the first thread can assign false to b, and then the second thread can finally assign true to c. The last step then resumes from ℓ_2 and ℓ_4, where the second thread performs the final assignment to variable b.

Preemption of system behaviors is very important for reactive systems. It is therefore very convenient that languages like Esterel and Quartz offer `abort` and `suspend` statements to explicitly express preemption. The meaning of these statements is as follows: A statement S which is enclosed by an `abort` block is immediately terminated when the given condition σ holds. Similarly, the `suspend` statement freezes the control flow in a statement S when σ holds. Thereby, two kinds of preemption must be distinguished: strong (default) and weak (indicated by keyword `weak`) preemption. While strong preemption deactivates both the control and data flow of the current step, weak preemption only deactivates the control flow, but retains the data flow of the current macro step (this concept is sometimes also called 'run-to-completion'). The immediate variants check for preemption already at starting time, while the default is to check preemption only after starting time.

Modular design is supported by the declaration of modules in the source code and by calling these modules in statements. Any statement can be encapsulated in a

module, which further declares a set of input and output signals for interaction with its context statement. There are no restrictions for module calls, so that modules can be instantiated in every statement. In contrast to many other languages, a module instantiation can also be part of sequences or conditionals (and is therefore not restricted to be called as additional thread). Furthermore, it can be located in any abortion or suspension context, which possibly preempts its execution.

The pioneer of the class of imperative synchronous languages is Esterel. As the description above suggests, both Esterel and Quartz share many principles, but there are also some subtle differences: First, the concept of pure and valued signals in Esterel has been generalized to *event* and *memorized* variables in Quartz as presented above. In particular, there is no distinction between `if` and `present` statements, and the reaction to absence considers arbitrary variables and not just signals. Another important difference is causality: while Esterel usually sees the statements $S_1; S_2$ as a real sequence, where the backward flow of information from S_2 to S_1 is forbidden, Quartz has a more relaxed definition of that. As long as S_1 and S_2 are executed in the same macro step, information can be arbitrarily exchanged, as the example in Fig. 2.3 illustrates. In consequence, more programs are considered to be causally correct. In contrast to Esterel, Quartz offers also the delayed assignments $\texttt{next}(x) = \tau$ that are convenient to describe hardware designs (while Esterel makes use of a `pre` operator to refer to the previous value of a signal).

2.3 Compilation

2.3.1 Intermediate Representation by Guarded Actions

Having presented the syntax and semantics of the synchronous language Quartz in the previous section, we now describe how it is compiled to hardware and software systems. As usual for compilers, we thereby make use of an internal representation of the program that can be used for analysis and the later synthesis. Especially in the design of embedded systems, where hardware-software partitioning and target platforms are design decisions that are frequently changed, persistent intermediate results stored in an internal representation are very important. It is thereby natural to distinguish between *compilation and synthesis*: Compilation is the translation of the Quartz program into the internal representation, and synthesis is the translation from the internal representation to traditional hardware or software descriptions.

In our Averest system, which is a framework for the synthesis and verification of Quartz programs, we have chosen *synchronous guarded actions* as internal representation that we called Averest Intermediate Format (AIF). Synchronous guarded actions are in the spirit of traditional guarded commands [15, 18, 27, 30] but follow the synchronous MoC. The Boolean condition γ of a guarded action $\langle \gamma \Rightarrow \mathscr{C} \rangle$ is called the guard and the atomic statement $\mathscr{C}$ is called the action of the guarded action. According to the previous section, atomic statements are

essentially the assignments of Quartz, i.e., the guarded actions have either the form $\gamma \Rightarrow x = \tau$ (for an immediate assignment) or $\gamma \Rightarrow \mathsf{next}(x) = \tau$ (for a delayed assignment).

The intuition behind such a guarded action $\gamma \Rightarrow \mathscr{C}$ is that the action $\mathscr{C}$ is executed in every macro step where the condition γ is satisfied. Guarded actions may be viewed as a simple programming language like Unity [15] in that every guarded action runs as a separate process in parallel to the other guarded actions. This process observes its guard γ in each step and executes $\mathscr{C}$ if the guard holds. The *semantics of synchronous guarded actions* is simply defined as follows: In every macro step, all guards are simultaneously checked. If a guard is true, its action is immediately executed: immediate assignments instantaneously transfer the computed value to the left-hand side of the assignment, while the delayed assignments defer the transfer to the next macro step. As there may be interdependencies between actions and trigger conditions, the actions must be executed according to their data dependencies. Similar to the Quartz program, the AIF description handles the *reaction to absence* implicitly: If no action has determined the value of the variable in the current macro step (obviously, this is the case iff the guards of all immediate assignments in the current step and the guards of all delayed assignments in the preceding step of a variable are false), then its value is determined by the reaction to absence according to its storage mode: *Event* variables are reset to their default values (like wires in hardware circuits), while *memorized* variables store their previous values (like registers in hardware circuits). Future versions of Quartz will contain clocked variables that are absent if not explicitly assigned in a step, i.e., the reaction to absence will not provide any value for them.

We are convinced that this representation of the behavior is exactly at *the right level of abstraction* for an intermediate code format, since guarded actions provide a good balance between (1) removal of complexity from the source code level and (2) the independence of a specific synthesis target. The semantics of complex control flow statements can be completely encoded by the guarded actions, so that the subsequent analysis, optimization, and synthesis steps become much simpler: Due to their very simple structure, efficient translation to both software and hardware is efficiently possible from guarded actions.

In general, programs written in all synchronous languages can be translated to synchronous guarded actions [8–10]. While this translation is straightforward for data flow languages such as Lustre, more effort is needed for imperative languages such as Quartz and Esterel. There, the translation has to extract all actions of the program and to compute for each of them a trigger condition according to the program. In the rest of this section, we describe the basics of the translation from Quartz programs to guarded actions. For a better understanding, we neglect local variables and module calls and rather describe a simple, but incomplete translation in this section. More details of the translation are given in [11].

In the following, we first discuss the distinction of surface and depth of a program in Sect. 2.3.2. Based on this distinction, we present the compilation of the control flow in Sect. 2.3.3, before we focus on the data flow in Sect. 2.3.4. Finally, we consider the additional problems due to local variables in Sect. 2.3.5.

2.3.2 Surface and Depth

A key to the compilation of synchronous programs is the distinction between the *surface and depth* of a program: Intuitively, the surface consists of the micro steps that are executed when the program is started, i.e., all the parts that are executed before reaching the first `pause` statements. The depth contains the statements that are executed when the program resumes execution after the first macro step, i.e., when the control is already inside the program and proceeds with its execution. It is important to note that surface and depth may overlap, since `pause` statements may be conditionally executed. Consider the example shown in Fig. 2.4a: while the action $x = 1$ is only in the surface and the action $z = 1$ is only in the depth, the action $y = 1$ is both in the depth (Fig. 2.4b) and the surface (Fig. 2.4c) of the sequence.

The example shown in Fig. 2.5 illustrates the necessity of distinguishing between the surface and the depth for the compilation. The compilation should compute for the data flow of a statement S guarded actions of the form $\langle \gamma \Rightarrow \mathscr{C} \rangle$, where $\mathscr{C}$ is a Quartz assignment, which is executed if and only if the condition γ holds. One may think that the set of guarded actions for the data flow can be computed by a simple recursive traversal over the program structure, which keeps track of the precondition leading to the current position. However, this is not the case, as the example in Fig. 2.5 illustrates. Since the abortion is not an immediate one, the assignment $a = \mathsf{true}$ will never be aborted, while the assignment $b = \mathsf{true}$ will be aborted if i holds. Now, assume we would first compute guarded actions for the body of the abortion statement and would then replace each guard φ by $\varphi \wedge \neg i$ to implement the abortion. For the variable a, this incorrect approach would derive two

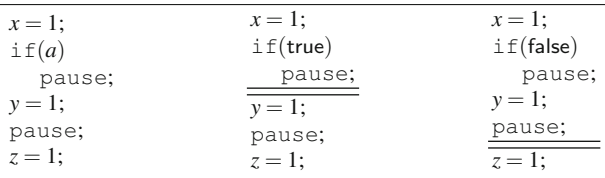


Fig. 2.4 Overlapping surface and depth: (**a**) Source code. (**b**) Case $a = \mathsf{true}$. (**c**) Case $a = \mathsf{false}$

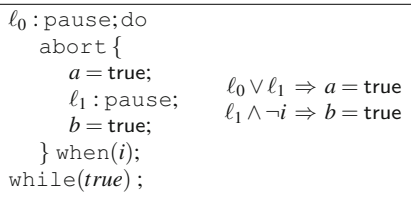


Fig. 2.5 Using surface and depth for the compilation

guarded actions $\ell_0 \wedge \neg i \Rightarrow a = \mathsf{true}$ and $\ell_1 \wedge \neg i \Rightarrow a = \mathsf{true}$. However, this is obviously wrong since now both assignments $a = \mathsf{true}$ and $b = \mathsf{true}$ are aborted which is not the semantics of the program.

The example shows that we have to distinguish between the guarded actions of the surface and the depth of a statement since these must be treated differently by the preemption statements. If we store these actions in two different sets, then we can simply add the conjunct $\neg\sigma$ to the guards of the actions of the depth, while leaving the guards of the actions of the surface unchanged. For this reason, we have to compute guarded actions for the surface and the depth in two different sets.

2.3.3 Compilation of the Control Flow

The control flow of a synchronous program may only rest at its control flow locations. Hence, it is sufficient to describe all situations where the control flow can move from the set of currently active locations to the set of locations that are active at the next point of time. The control flow can therefore be described by actions of the form $\langle \gamma \Rightarrow \mathsf{next}(\ell) = \mathsf{true} \rangle$, where ℓ is a Boolean event variable modeling the control flow location and γ is a condition that is responsible for moving the control flow at the next point of time to location ℓ. Since a location is represented by an event variable, its reaction to absence resets it to false whenever no guarded action explicitly sets it.

Thus, the compiler has to extract from the synchronous program for every label ℓ a set of trigger conditions $\phi_1^\ell, \ldots, \phi_n^\ell$ to determine whether this label ℓ has to be set in the following step. Then, these conditions can be encoded as guarded actions $\langle \phi_1^\ell \Rightarrow \mathsf{next}(\ell) = \mathsf{true} \rangle, \ldots, \langle \phi_n^\ell \Rightarrow \mathsf{next}(\ell) = \mathsf{true} \rangle$. The whole control flow is then just the union of all sets for all labels. Hence, in the following, we describe how to determine the activation conditions ϕ_i^ℓ for each label ℓ of the program.

The compilation is implemented as a bottom-up procedure that extracts the control flow by a recursive traversal over the program structure: For example, for a loop, we first compile the loop body and then add the loop behavior. While descending in the recursion, we determine the following conditions and forward them to the compilation of the substatements of a given statement S:

- $\mathsf{strt}(S)$ is the current activation condition. It holds iff S is started in the current macro step.
- $\mathsf{abrt}(S)$ is the disjunction of the guards of all abort blocks which contain S. Hence, the condition holds iff S should be currently aborted.
- $\mathsf{susp}(S)$ similarly describes the suspension context: if the predicate holds, S will be suspended. Thereby, $\mathsf{abrt}(S)$ has a higher priority, i.e., if both $\mathsf{abrt}(S)$ and $\mathsf{susp}(S)$ hold, then the abortion takes place.

The compilation of S returns the following control flow predicates [41], which are used for the compilation of the surrounding compound statement.

- $\mathsf{inst}(S)$ holds iff the execution of S is currently instantaneous. This condition depends on inputs so that we compute an expression $\mathsf{inst}(S)$ depending on the current values of input, local, and output variables. In general, $\mathsf{inst}(S)$ cannot depend on the locations of S since it is checked whether the control flows through S without being caught in S. Hence, it is assumed that S is currently not active.
- $\mathsf{insd}(S)$ is the disjunction of the labels in statement S. Therefore, $\mathsf{insd}(S)$ holds at some point of time iff the control flow is currently at some location inside S, i.e., if S is active. Thus, instantaneous statements are never active, since the control flow cannot rest anywhere inside.
- $\mathsf{term}(S)$ describes all conditions where the control flow is currently somewhere inside S and wants to leave S voluntarily. Note, however, that the control flow might still be in S at the next point of time, since S may be (re)entered at the same time, e.g., by a surrounding loop statement. The expression $\mathsf{term}(S)$ therefore depends on input, local, output, and location variables. $\mathsf{term}(S)$ is false whenever the control flow is currently not inside S. In particular, $\mathsf{term}(S)$ is false for the instantaneous atomic statements.

The control flow predicates refer either to the surface or to the depth of a statement. As it will be obvious in the following, the surface uses $\mathsf{strt}(S)$ and $\mathsf{inst}(S)$, while the depth depends on $\mathsf{abrt}(S)$, $\mathsf{susp}(S)$, $\mathsf{insd}(S)$ and $\mathsf{term}(S)$. Hence, we can divide the compilation of each statement into two functions: one compiles its surface and the other one compiles its depth.

After these introductory explanations, we can now present the general structure of the compilation algorithm (see Fig. 2.6). The compilation of a system consisting of a statement S is initially started by the function $\mathsf{ControlFlow}(\mathrm{st}, S)$, which splits the task into surface and depth parts. Abort and suspend conditions for the depth are initially set to false, since there is no preemption context at this stage.

It remains to show how the surface and the depth of each statement are compiled. Thereby, we forward the previously determined control flow context for a statement S by the Boolean values $\mathrm{st} = \mathsf{strt}(S)$, $\mathrm{ab} = \mathsf{abrt}(S)$, and $\mathrm{sp} = \mathsf{susp}(S)$, while the result contains the values of the predicates $I = \mathsf{inst}(S)$, $A = \mathsf{insd}(S)$, and $T = \mathsf{term}(S)$.

Let us start with the assignments of the program. Since they do not contribute to the control flow, no guarded actions are derived from them. The computation of the control flow predicates is also very simple: An action $\mathscr{C}$ is always instantaneous $\mathsf{inst}(\mathscr{C}) = \mathsf{true}$, never active ($\mathsf{insd}(\mathscr{C}) = \mathsf{false}$), and never terminates (since the control flow cannot rest inside $\mathscr{C}$; see definitions above).

The pause statement is interesting, since it is the only one that creates actions for the control flow. The surface part of the compilation detects when a label is activated: each time we hit a ℓ : `pause` statement, we take the activation condition computed so far and take this for the creation of a new guarded action setting ℓ. The label ℓ can be also activated later in the depth. This is the case if the control is currently at this label and the outer context requests the suspension. The

```
fun ControlFlow(st, S)
    (I, C^s) = CtrlSurface(st, S);
    (A, T, C^d) = CtrlDepth(false, false, S);
    return(C^s ∪ C^d)

fun CtrlSurface(st, S)
  switch(S)
    case [ℓ : pause]
        return(false, {st ⇒ next(ℓ) = true})
    case [if(σ) S1 else S2]
        (I1, C1^s) = CtrlSurface(st ∧ σ, S1);
        (I2, C2^s) = CtrlSurface(st ∧ ¬σ, S2);
        return(I1 ∧ σ ∨ I2 ∧ ¬σ, C1^s ∪ C2^s)
    case [S1; S2]
        (I1, C1^s) = CtrlSurface(st, S1);
        (I2, C2^s) = CtrlSurface(st ∧ I1, S2);
        return(I1 ∧ I2, C1^s ∪ C2^s)
    case [abort S1 when(σ)]
        returnCtrlSurface(st, S1)
        ⋮

fun CtrlDepth(ab, sp, S)
  switch(S)
    case [ℓ : pause]
        return(ℓ, ℓ, {ℓ ∧ sp ⇒ next(ℓ) = true})
    case [if(σ) S1 else S2]
        (A1, T1, C1^d) = CtrlDepth(ab, sp, S1);
        (A2, T2, C2^d) = CtrlDepth(ab, sp, S2);
        return(A1 ∨ A2, T1 ∨ T2, C1^d ∪ C2^d)
    case [S1; S2]
        (A1, T1, C1^d) = CtrlDepth(ab, sp, S1);
        st2 = T1 ∧ ¬(sp ∨ ab);
        (I2, C2^s) = CtrlSurface(st2, S2);
        (A2, T2, C2^d) = CtrlDepth(ab, sp, S2);
        return(A1 ∨ A2, T1 ∧ I2 ∨ T2, C1^d ∪ C2^s ∪ C2^d)
    case [abort S1 when(σ)]
        (A1, T1, C1^d) = CtrlDepth(ab ∨ σ, sp, S1);
        return(A1, T1 ∨ A1 ∧ σ, C1^d)
        ⋮
```

Fig. 2.6 Compiling the control flow (excerpt)

computation of the control flow predicates reveals no surprises: ℓ : `pause` always needs time $\mathsf{inst}() = \mathsf{false}$, it is active if the control flow is currently at label ℓ ($\mathsf{insd}() = \ell$) and it terminates whenever it is active ($\mathsf{term}() = \ell$).

Let us now consider a compound statement like a conditional statement. For the surface, we update the activation condition by adding the guard. Then, the substatements are compiled and the results are merged straightforwardly. The parallel statement (not shown in the figure) is compiled similarly. A bit more interesting is the compilation of the sequence. As already mentioned above, it implements the stepwise traversal of the synchronous program. This is accomplished by the surface calls in the depth. A similar behavior can also be found in the functions for the compilation of the loops. Preemption is also rather simple: since the abortion condition is usually not checked when entering the abort statement, it does not have any influence and can be neglected for the compilation of the surface. In the depth, the abort condition is just appended to the previous one. Finally, suspension is compiled similarly.

The compilation of the control flow works in a linear pass over the syntax tree of the program and generates a set of guarded actions of a linear size with respect to the size of the given program.

2.3.4 Compilation of the Data Flow

Figure 2.7 shows some of the functions which compute the data flow for a given Quartz statement where we make use of a virtual boot label ℓ_0. The surface actions can be executed in the current step, while the actions of the depth are enabled by the resumption of the control flow that already rests somewhere in the statement. For this reason, we do not need a precondition as argument for the depth data flow compilation since the precondition is the active current control flow location itself. The compilation of the surface of a basic statement should be clear: we take the so-far computed precondition st as the guard for the atomic action. The depth variants do not create any actions since statements without control flow locations do not have depth actions.

For the conditional statement, we simply add σ or its negation to the precondition to start the corresponding substatement. As the control flow can rest in one of the branches of an if-statement, it can be resumed from any of these branches. In the depth, we therefore simply take the "union" of the two computations of the depth actions.

fun DataFlow(S)
  $\mathscr{D}^{\mathsf{s}} = \mathsf{DataSurface}(\ell_0, S)$;
  $\mathscr{D}^{\mathsf{d}} = \mathsf{DataDepth}(S)$;
  return$(\mathscr{D}^{\mathsf{s}} \cup \mathscr{D}^{\mathsf{d}})$

fun DataSurface(st, S)
 switch(S)
  case $[x = \tau]$
    return$\{\mathrm{st} \Rightarrow x = \tau\}$
  case $[\mathsf{next}(x) = \tau]$
    return$\{\mathrm{st} \Rightarrow \mathsf{next}(x) = \tau\}$
  case $[\texttt{if}(\sigma)\ S_1 \texttt{else}\ S_2]$
    $\mathscr{D}_1^{\mathsf{s}} = \mathsf{DataSurface}(\mathrm{st} \wedge \sigma, S_1)$
    $\mathscr{D}_2^{\mathsf{s}} = \mathsf{DataSurface}(\mathrm{st} \wedge \neg\sigma, S_2)$
    return$(\mathscr{D}_1^{\mathsf{s}} \cup \mathscr{D}_2^{\mathsf{s}})$
  case $[S_1; S_2]$
    $\mathscr{D}_1^{\mathsf{s}} = \mathsf{DataSurface}(\mathrm{st}, S_1)$
    $\mathscr{D}_2^{\mathsf{s}} = \mathsf{DataSurface}(\mathrm{st} \wedge \mathsf{inst}(S_1), S_2)$
    return$(\mathscr{D}_1^{\mathsf{s}} \cup \mathscr{D}_2^{\mathsf{s}})$
  case $[\texttt{abort}\ S_1\ \texttt{when}(\sigma)]$
    return DataSurface(st, S_1)
  case $[\texttt{weak abort}\ S_1\ \texttt{when}(\sigma)]$
    return DataSurface(st, S_1)
    ⋮

fun DataDepth(S) =
 switch(S)
  case $[x = \tau]$
    return$\{\}$
  case $[\mathsf{next}(x) = \tau]$
    return$\{\}$
  case $[\texttt{if}(\sigma)\ S_1 \texttt{else}\ S_2]$
    $\mathscr{D}_1^{\mathsf{d}} = \mathsf{DataDepth}(S_1)$
    $\mathscr{D}_2^{\mathsf{d}} = \mathsf{DataDepth}(S_2)$
    return$(\mathscr{D}_1^{\mathsf{d}} \cup \mathscr{D}_2^{\mathsf{d}})$
  case $[S_1; S_2]$
    $\mathscr{D}_1^{\mathsf{d}} = \mathsf{DataDepth}(S_1)$
    $\mathscr{D}_2^{\mathsf{s}} = \mathsf{DataSurface}(\mathsf{term}(S_1), S_2)$
    $\mathscr{D}_2^{\mathsf{d}} = \mathsf{DataDepth}(S_2)$
    return$(\mathscr{D}_1^{\mathsf{d}} \cup \mathscr{D}_2^{\mathsf{s}} \cup \mathscr{D}_2^{\mathsf{d}})$
  case $[\texttt{abort}\ S_1\ \texttt{when}(\sigma)]$
    return
      $\{\gamma \wedge \neg\sigma \Rightarrow \mathscr{C} \mid (\gamma \Rightarrow \mathscr{C}) \in \mathsf{DataDepth}(S_1)\}$
  case $[\texttt{weak abort}\ S_1\ \texttt{when}(\sigma)]$
    return DataDepth(S_1)
    ⋮

Fig. 2.7 Compiling the data flow (excerpt)

According to the semantics of a sequence, we first execute S_1. If the execution of S_1 is instantaneous, then we also execute S_2 in the same macro step. Hence, the precondition for the surface actions of S_2 is $\varphi \wedge \mathsf{inst}(S_1)$. The preconditions of the substatements of a parallel statement are simply the preconditions of the parallel statement.

In the depth, the control flow can rest in either one of the substatements S_1 or S_2 of a sequence $S_1; S_2$, and hence, we can resume it from either S_1 or S_2. If the control flow is resumed from somewhere inside S_1, and S_1 terminates, then also the surface actions of S_2 are executed in the depth of the sequence. Note that the computation of the depth of a sequence $S_1; S_2$ leads to the computation of the surface actions of S_2 as in the computation of the control flow in the previous section.

As delayed abortions are ignored at starting time of a delayed abort statement, we can ignore them for the computation of the surface actions. Weak preemption statements can be also ignored for the computation of the depth actions, since even if the abortion takes place, all actions remain enabled due to the weak preemption. For the depth of strong abortion statements, we add a conjunct $\neg\sigma$ to the guards of all actions to disable them in case σ holds.

Obviously, the compilation of the control flow (see Fig. 2.6) and the compilation of the data flow (see Fig. 2.7) can be merged into a single set of functions that simultaneously compile both parts of the program. Since the guards of the actions for the data flow refer to the control flow predicates, this approach simplifies the implementation. The result is an algorithm which runs in time $O(|S|^2)$, since $\mathsf{DataSurface}(S,)$ runs in $O(|S|)$ and $\mathsf{DataDepth}(S)$ in $O(|S|^2)$. The reason for the quadratic blow-up is that sequences and loops necessarily have to generate copies of surfaces of their substatements.

2.3.5 Local Variables and Schizophrenia

The characteristic property of local variables is their limited scope. In the context of synchrony, which groups a number of micro steps into an instantaneous macro step, a limited scope which does not match with the macro steps may cause problems. In particular, this is the case if a local declaration is left and reentered within the same macro step, e.g., when a local declaration is nested within loop statements. In such a problematic macro step, the micro steps must then refer to the right incarnation of the local variable since its incarnations in the old and the new scope may have different values in one macro step.

Figure 2.8a shows a simple example. The local variable x, which is declared in the loop body, is referenced at the beginning and at the end of the loop. In the second step of the program, when it resumes from the label ℓ_1, all actions are executed, but they refer to two different incarnations of x: While the assignment to x is made to the old variable in the depth, the if-statement checks the value of the new incarnation which is false.

While software synthesis of sequential programs can solve this problem simply by shadowing the incarnations of the old scope, this is not possible for the

a

```
do {
   bool x;
   if(x) y = 1;
   ℓ1 : pause;
   x = true;
} while(true);
```

b

```
do {
   bool x;
   if(x) y = 1;
   if(a) ℓ1 : pause;
   x = true;
   if(¬a) ℓ2 : pause;
} while(true);
```

Fig. 2.8 Schizophrenic Quartz programs

synchronous MoC, since each variable has exactly one value per macro step. Therefore, we have to generate a copy of the locally declared variable and map the actions of the program to the corresponding copy in the intermediate code. Furthermore, we have to create additional actions in the intermediate code that link the copies so that the value of the new incarnation at the beginning of the scope is eventually transported to the old one, which is used in the rest of the scope.

However, the problem can be even worse: first, whereas in the previous example each statement always referred to the same incarnation (either the old or the new one), the general case is more complicated as can be seen in Fig. 2.8b. The statements between the two pause statements are sometimes in the context of the old and sometimes in the context of the new incarnation. Therefore, these statements are usually called *schizophrenic* in the synchronous languages community [4]. Second, there can be several reincarnations of a local variable, since the scope can be reentered more than once. In general, the number of loops, which are nested around a local variable declaration determines an upper bound on the number of possible reincarnations.

A challenging Quartz program containing local variables is shown on the left-hand side of Fig. 2.9. The right-hand side of the same figure shows the corresponding control flow graph. The circle nodes of this graph are control flow states that are labeled with those location variables that are currently active (including the start location ℓ_0). Besides these control flow states, there are two other kinds of nodes: boxes contain actions that are executed when an arc toward this node is traversed, while the diamonds represent branches that influence the following computations. The outgoing arcs of such a node correspond to the *then* (solid) and *else* (dashed) branch of the condition. For example, if the program is executed from state ℓ_1 and we have $\neg k \wedge j \wedge \neg i$, then we execute the two action boxes beneath control state ℓ_1 and additionally the one below condition node j.

As can be seen, the condition $k \wedge j \wedge \neg i$ executes all possible action nodes while traversing from control node ℓ_1 to itself. The first action node belongs to the depth of all local declarations, the second one (re)enters the local declaration of c, but remains inside the local declarations of b and a. A new incarnation c_3 is thereby created. The node below condition node k (re)enters the local declarations of b and c, but remains in the one of a. Hence, it creates new incarnations b_2 and c_2 of b

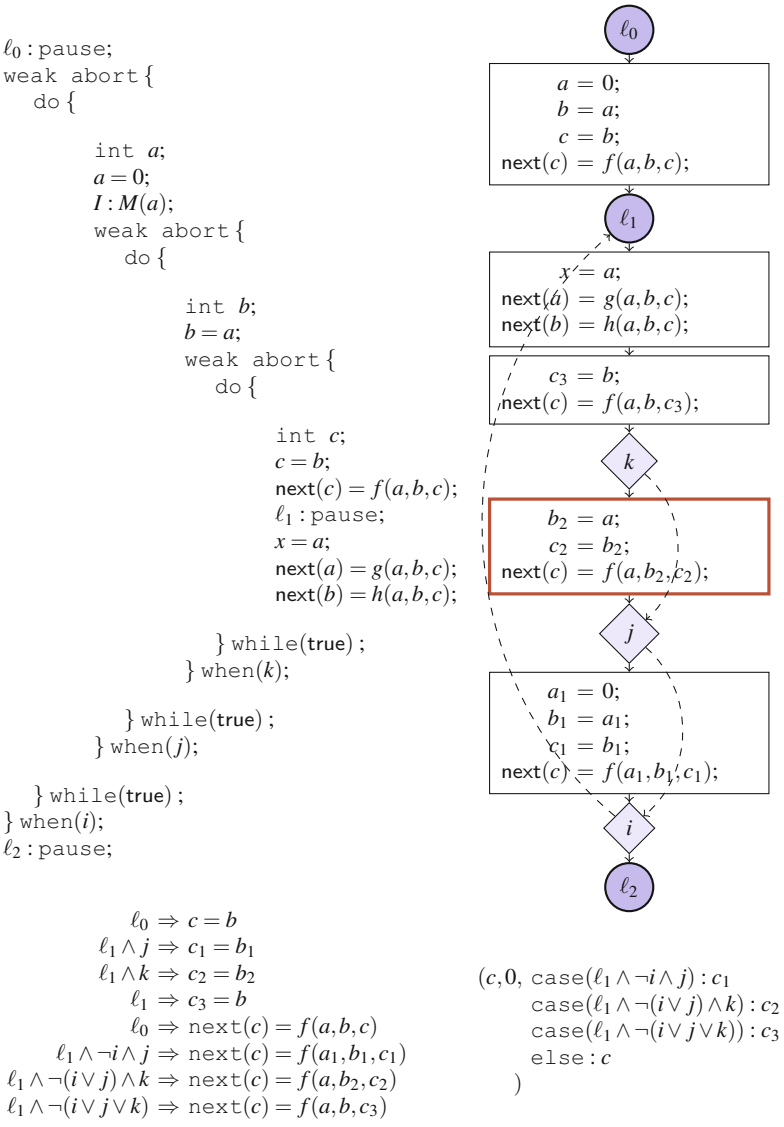


Fig. 2.9 Local declarations with multiple simultaneous reincarnations

and c, respectively. Finally, the remaining node (re)enters all local declarations and therefore generates three incarnations a_1, b_1, and c_1. Note that these four action boxes can be executed at the same point of time, and therefore, the reincarnations a_1, b_1, c_1, b_2, c_2, and c_3 may all exist in one macro step.

Several solutions have been proposed for the solution of schizophrenic statements [4, 35, 41, 49, 54]. Our Quartz compiler carefully distinguishes between the different surfaces of the same schizophrenic local variable and creates fresh incarnations for each one. Technically, this is handled by adding a counter to the compilation functions (Figs. 2.6 and 2.7), which counts how often such a surface has already been entered in the same reaction. Giving all technical details is beyond the scope of this chapter. The interested reader is referred to [11, 47], where the complete solution is described.

2.4 Semantic Analysis

The synchronous MoC abstracts from the execution order within a reaction and postulates that all actions are executed in zero time – at least in the programmer's view. In practice, this means that all actions must be executed according to their data dependencies in order to keep the illusion of zero time execution for the programmer. However, it could happen that there is no such execution order since there are cyclic data dependencies. These so-called causality cycles occur if the input for an action is instantaneously modified by the output of this action or others that were triggered by its output. From the practical side, cyclic programs are rather rare, but they can appear and must therefore be handled by compilers. As a consequence, *causality analysis* [4, 7, 13, 24, 34, 45, 48, 52] must be performed to detect critical cyclic programs.

In general, cyclic programs might have no behavior (loss of reactivity), more than one behavior (loss of determinism), or a unique behavior. However, having a unique behavior is not sufficient for causality, since there are programs whose unique behavior can only be found by guessing. For this reason, causality analysis checks whether a program has a unique behavior that can furthermore be constructively determined by the operational semantics of the program. To this end, the causality analysis starts with known input variables and yet unknown local/output variables. Then, it determines the micro steps of a macro step that can be executed with this incomplete knowledge of the current variables' values. The execution of these micro steps may reveal the values of some further local/output variables of this macro step so that further micro steps can be executed after this round. If all variables became finally known by this fixpoint iteration, the program is a constructive one with a unique behavior.

While causality analysis may appear to be a special problem for synchronous languages, a closer look at the problem reveals that there are many equivalent problems: (1) Shiple [51–53] proved the equivalence to Brzozowski and Seger's timing analysis in the up-bounded inertial delay model [13]. This means that a circuit derived from a cyclic equation system will finally stabilize for arbitrary gate delays iff the equation system is causally correct. (2) Berry pointed out that causality analysis is equivalent to theorem proving in intuitionistic (constructive) propositional logic and introduced the notion of constructive circuits [5]. (3) The problem is also equivalent to type-checking in functional programs due to the

Curry-Howard isomorphism [26]. (4) Finally, Edwards reformulates the problem as the existence of dynamic schedules for the execution of mutually dependent threads of micro steps [20]. Hence, causality analysis is a fundamental analysis that has already found many applications in computer science.

We will not discuss the details in this chapter. The interested reader is referred to [45], where all details about the causality analysis performed in the context of Quartz can be found.

2.5 Synthesis

Before presenting the synthesis procedures, we first recall our overall design flow that determines the context of the compilation procedure. As we target the design of embedded systems, where hardware-software partitioning and target platforms are design decisions that are frequently changed, persistent intermediate results in a well-defined and robust format are welcome. In our Averest system, we basically split the design flow into two steps, which are bridged by the AIF. This intermediate format captures the system behavior in terms of synchronous guarded actions. Hence, complex control flow statements need no longer be considered. We refer to *compilation* as the translation of source code into AIF, while *synthesis* means the translation from AIF to the final target code, which may be based on a different MoC.

Figure 2.10 shows two approaches of generating target code from a set of Quartz modules. *Modular compilation*, which is shown on the left-hand side, translates each Quartz module to a corresponding AIF module. Then, these modules are linked on the intermediate level before the whole system is synthesized to target code. *Modular synthesis*, which is shown on the right-hand side, translates each Quartz module to a corresponding AIF module, which is subsequently synthesized to a target code module. Linking is then deferred to the target level. While modular synthesis simplifies the compilation (since all translation processes have to consider only a module of the system), it puts the burden on the run-time platform or the linker which has to organize the interaction of the target modules correctly.

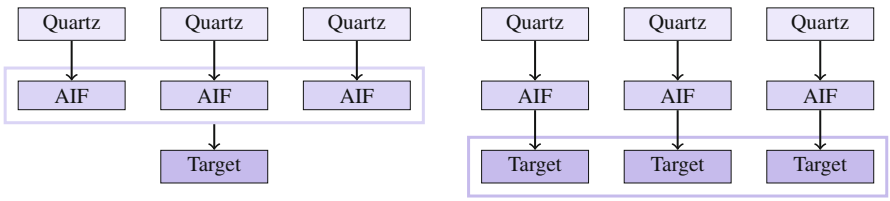


Fig. 2.10 Modular compilation and modular synthesis of Quartz programs

From our intermediate representation of guarded actions, many synthesis targets can be thought of. In the following, we sketch the translation to different targets; first to symbolic transition systems, which are suitable for formal verification of program properties by symbolic model checking, second to digital hardware circuits for hardware synthesis, third the translation to SystemC code, which can be used for an integrated simulation of the system, and finally an automaton-based sequential software synthesis.

2.5.1 Symbolic Model Checking

For symbolic model checking, the system generally needs to be represented by a transition system. This basically consists of a triple $(\mathscr{S}, \mathscr{I}, \mathscr{T})$ with a set of states $\mathscr{S}$, initial states $\mathscr{I} \subseteq \mathscr{S}$, and a transition relation $\mathscr{T} \subseteq \mathscr{S} \times \mathscr{S}$. Each state s is a mapping from variables to values, i.e., s assigns to each variable a value of its domain. As we aim for a symbolic description, we describe the initial states and the transition relation by propositional formulas $\Phi_{\mathscr{I}}$ and $\Phi_{\mathscr{T}}$, which are their characteristic functions.

For the presentation of the translation, assume that our intermediate representation contains immediate and delayed actions for each variable x of the following form:

$$(\gamma_1, \mathtt{x} = \tau_1), \quad \ldots, (\gamma_p, \mathtt{x} = \tau_p)$$
$$(\chi_1, \mathtt{next(x)} = \pi_1), \ldots, (\chi_q, \mathtt{next(x)} = \pi_q)$$

Figure 2.11 sketches the translation of the immediate and delayed actions writing variable x to clauses used for the description of a symbolic transition system.

The construction of a transition system is quite straightforward: The initial value of a variable x can only be determined by its immediate actions. Hence, if one of the guards γ_i of the immediate actions holds, the corresponding immediate assignment defines the value of x. If none of the guards γ_i should hold, the initial value of x is determined by its default value (which is determined by the semantics, e.g., false for Boolean variables and 0 for numeric ones).

The transition relation determines which states can be connected by a transition, i.e., which values the variables may have at the next point of time given values at the current point of time. To this end, the transition relation sets up constraints for the values at the current and the next point of time: First, also the immediate assignments have to be respected for the current point of time, i.e., whenever a guard γ_i of the immediate actions holds, the corresponding immediate assignment defines the current value of x. If one of the guards χ_i of the delayed assignments hold at the current point of time, the next value of x is determined by the corresponding delayed assignment. Finally, if the next value of x is not determined by an action, i.e., none of the guards γ_i of the immediate assignments hold at the next point of time and neither holds one of the guards χ_i at the current point of time, then the next value of x is determined by the reaction to absence.

$$\mathsf{Init}_x :\equiv \left(\begin{array}{l} \bigwedge_{j=1}^{p} (\gamma_j \rightarrow x = \tau_j) \wedge \\ \\ \left(\bigwedge_{j=1}^{p} \neg\gamma_j \right) \rightarrow x = \mathsf{Default}(x) \end{array} \right)$$

$$\mathsf{Trans}_x :\equiv \left(\begin{array}{l} \bigwedge_{j=1}^{p} (\gamma_j \rightarrow x = \tau_j) \wedge \\ \\ \bigwedge_{j=1}^{q} (\chi_j \rightarrow \mathsf{next}(x) = \pi_j) \wedge \\ \\ \mathsf{next}(\bigwedge_{j=1}^{p} \neg\gamma_j) \wedge \left(\bigwedge_{j=1}^{q} \neg\chi_j \right) \rightarrow \mathsf{next}(x) = \mathsf{Abs}(x) \end{array} \right)$$

$$\mathsf{Abs}(x) := \begin{cases} \mathsf{Default}(x) & : \text{if } x \text{ is an event variable} \\ x & : \text{if } x \text{ is a memorized variable} \end{cases}$$

Fig. 2.11 Transition relation for x

The definitions given in Fig. 2.11 can be literally used to define input files for symbolic model checkers. Causality problems do not bother in this translation, and also write conflicts will show up as deadend states in the transition diagram and can be checked this way by symbolic model checking.

2.5.2 Circuit Synthesis

The transition relation shown in Fig. 2.11 of the previous section can be modified so that both the initial condition and the transition relation become equation systems provided that there are no write conflicts between the actions of any variable x. To explain the construction of the equations, we consider again any variable x with the following guarded actions:

$$\begin{array}{l} (\gamma_1, \mathtt{x} = \tau_1), \qquad\qquad \ldots, (\gamma_p, \mathtt{x} = \tau_p) \\ (\chi_1, \mathtt{next(x)} = \pi_1), \ldots, (\chi_q, \mathtt{next(x)} = \pi_q) \end{array}$$

The equations for x are shown in Fig. 2.12 where an additional carrier variable x' is used. This carrier variable x' has initially the default value of x, and its value at any next point of time is determined by the delayed assignments of x. If none of the delayed assignments is enabled, the reaction to absence is applied, i.e., $\mathsf{next}(x') = \mathsf{Abs}(x)$ will hold.

For the variable x itself, we introduce only an immediate equation where the current value of x is defined by the immediate assignments to it, and if none of them is enabled, we use the current value of the carrier variable to define x.

One can prove that x has this way the correct value for any point of time by induction over time: At the initial point of time, one of the immediate assignments will determine the initial value of x if one is enabled, which is equivalent to the initial condition of Fig. 2.11. If none of the γ_i holds, we have $x := x' := \mathsf{Default}(x)$ which is also the case for the initial condition of Fig. 2.11.

To determine the value of x at any later point of time, note first that again the immediate assignments can determine its value. If none of them is enabled, we have again $x := x'$. If we consider this equation from the previous point of time, it means $\mathsf{next}(x) = \mathsf{next}(x')$ so that we can see that x may be determined by delayed assignments that were enabled in the previous point of time. Finally, if none of these were enabled either, then the reaction to absence takes place which had already determined the value of x' so that $x := x'$ will define the value of x correctly also in this case.

Note that the equation system as given in Fig. 2.12 has only immediate equations for the output and local variables, while carrier variables are used to capture the delayed assignments. Since the carrier variables are not observable outside the module, they define the internal state together with the local variables of the program and the control flow labels of the pause statements. Note that these equation systems have exactly the form we assumed in the causality analysis described in Sect. 2.4.

Finally, we note that the actual synthesis of the equations is much more difficult due to the reincarnation of local variables. Since the reincarnations do only occur in the surface statements, and since surface statements can be executed in zero time, the behavior of the reincarnated variables can be described by simple immediate equations without carrier variables. However, the reaction to absence is more complicated for the local variable that remains in the depth of a local declaration statement: Here we have to determine which of the different surfaces has been the latest one executed that carries then its value to the depth. Furthermore, the delayed assignments to local variables in those reincarnated surfaces that are followed by further reincarnations have to be disabled. The overall procedure is quite difficult and is described in full detail in [43, 46].

$$\mathsf{init}(x') = \mathsf{Default}(x)$$

$$\mathsf{next}(x') = \begin{pmatrix} \textbf{case} \\ \quad \chi_1 : \pi_1; \\ \quad \vdots \\ \quad \chi_q : \pi_q; \\ \textbf{else}\ \mathsf{Abs}(x) \end{pmatrix} \qquad x = \begin{pmatrix} \textbf{case} \\ \quad \gamma_1 : \tau_1; \\ \quad \vdots \\ \quad \gamma_p : \tau_p; \\ \textbf{else}\ x' \end{pmatrix}$$

Fig. 2.12 Equation system for x

2.5.3 SystemC Simulation

The simulation semantics of SystemC is based on the discrete-event model of computation [14], where reactions of the system are triggered by events. All threads that are sensitive to a specified set of events are activated and produce new events during their execution. Updates of variables are not immediately visible, but become visible in the next delta cycle.

We start the translation by the definition of a global clock that ticks in each instant and drives all the computation. Thus, we require that the processed model is *endochronous* [22, 23], i.e., there is a signal which is present in all instants of the behavior and from which all other signals can be determined. In SystemC, this clock is implemented by a single `sc_clock` at the uppermost level, and all other components are connected to this clock. Hence, the translations of the macro steps of the synchronous program in SystemC are triggered by this clock, while the micro steps are triggered by signal changes in the delta cycles. For this reason, input and output variables of the synchronous program are mapped to input signals (`sc_in`) and output signals (`sc_out`) of SystemC of the corresponding type.

Additionally, we declare signals for all other clocks of the system. They are inputs since the clock constraints (as given by **`assume`**) do not give an operational description of the clocks, but can be only checked in the system. The clock calculus for Signal [1, 22, 23] or scheduler creation for CAOS [12] aim at creating exactly these schedulers which give an operational description of the clocks. Although not covered in the following, their result can be linked to the system description so that clocks are driven by the system itself.

The translation of the synchronous guarded actions to SystemC processes is however not as simple as one might expect. The basic idea is to map guarded actions to methods which are sensitive to the read variables so that the guarded action is re-evaluated each time one of the variables it depends on changes. For a *constructive* model, it is guaranteed that the simulation does not hang up in delta cycles.

The translation to SystemC must tackle the following two problems: (1) As SystemC does not allow a signal to have multiple drivers, all immediate and delayed actions must be grouped by their target variables, or equivalently, we can produce the equations as shown in the previous section. (2) The SystemC simulation semantics can lead to spurious updates of variables (in the AIF context), since threads are always triggered if some variables in the sensitivity list have been updated – even if they are changed once more in later delta cycles. As actions might be spuriously activated, it must be ensured that at least one action is activated in each instant, which sets the final value. Both problems are handled in a similar way as the translation to the transition system presented in the previous section: we create an additional variable `_carrier_x` for each variable x to record values from their delayed assignments and group all actions in the same way as for the transition system.

With these considerations, the translation of the immediate guarded actions $\langle \gamma \Rightarrow x = \tau_i \rangle$ is straightforward: We translate each group of actions into an asynchronous thread in SystemC, which is sensitive to all signals read by these actions (variables appearing in the guards γ_i or in the right-hand sides τ_i). Thereby, all actions are

```
void Module                                  void Module
  ::compute_x()                                ::compute_delayed_x()
{                                            {
  while(true) {                                while(true) {
    if(_clk_x.read() && γ1)                      if(ξ1)
      x.write(τ1);                                 _carrier_x.write(π1);
      ...                                          ...
    else if(_clk_x.read() && γn)                 else if(ξn)
      x.write(τn);                                 _carrier_x.write(πn);
    else if(_clk_x.read())                       else
      x.write(_carrier_x.read());                  _carrier_x.write(x.read());
    wait();                                      wait();
  }                                            }
}                                            }
```

Fig. 2.13 SystemC: Translation of immediate and delayed actions

implemented by an `if`-block except for the last one, which handles the case that no action fires. Since the immediate actions should become immediately visible, the new value can be immediately written to the variable with the help of a call to `x.write(...)`. Analogously, the evaluations of the guard γ_i and the right-hand side of the assignment τ_i make use of the `read` methods of the other signals. The left-hand side of Fig. 2.13 shows the general structure of such a thread.

Delayed actions $\langle \gamma \Rightarrow \texttt{next}(x) = \pi_j \rangle$ are handled differently: While the right-hand side is immediately evaluated, the assignment should only be visible in the following macro step and not yet in the current one. Hence, they do not take part in the fixpoint iteration. Therefore, we write their result to `_carrier_x` in a *clocked thread*, which is triggered by the master trigger Thereby, signals changed by the delayed actions do not affect the current fixpoint iteration and vice versa.

Figure 2.14 gives the SystemC code for the part that simulates the variables x. Similar to the Symbolic Model Verifier (SMV) translation, we abbreviate guards for reuse in different SystemC processes. Then, the translation of the immediate actions writing x is straightforward; they correspond to the first two cases in method `OuterQuartz::compute_x()`. The last case is responsible for setting x to its previous value if neither of the two immediate actions fires. As already stated above, we need to do this explicitly. To this end, the previous value of variable x is always stored in a separate carrier variable. For variables, which are only set by delayed actions, we can simplify the general scheme of Fig. 2.13. In this case, we can combine the two threads as Fig. 2.14 illustrates: we only need a single variable, which is set by this thread.

2.5.4 Automaton-Based Sequential Software Synthesis

In order to generate fast sequential code from synchronous programs, the Extended Finite-State Machine (EFSM) representation of the program is an ideal starting

```
void OuterQuartz
  ::compute_x()
{
  while(true) {
    if(_grd18)
      x.write(y.read());
    else if(_grd19)
      x.write(2*y.read());
    else if(_clk_x.read())
      x.write(_carrier_x.read());
    wait();
  }
}
```

```
void OuterQuartz
  ::compute_delayed_x()
{
  while(true) {
    _carrier_x.write(x.read());
    wait();
  }
}

void OuterQuartz
  ::compute_delayed_ell1()
{
  while(true) {
    if(_grd18)
      ell1.write(true);
    else
      ell1.write(false);
    wait();
  }
}
```

Fig. 2.14 SystemC code

point. EFSMs explicitly represent the state transition system of the control flow: each state s represents a subset $\mathsf{Labels}(s) \subseteq \mathscr{L}$ of the control flow labels, and edges between states are labeled with conditions that must be fulfilled to reach the target state from the source state. EFSMs are therefore a representation where the control flow of a program state is explicitly represented, while the data flow is still represented symbolically (while synchronous guarded actions represent control flow and data flow symbolically).

The guards of the guarded actions of the control flow are therefore translated to transition conditions of the EFSM's state transitions. The guarded actions of the data flow are first copied to each state of the EFSM and are then partially evaluated according to the values of the control flow labels in that EFSM state. Hence, in each macro step, the generated code will only consider a subset $D(s)$ of the guarded actions, which generally speeds up the execution (since many of them are only active in a small number of states).

Definition 1 (Extended Finite State Machine). An *Extended Finite-State Machine (EFSM)* is a tuple (S, s_0, T, D), where S is a set of states, $s_0 \in S$ is the initial state, and $T \subseteq (S * C * S)$ is a finite transition relation where C is the set of transition conditions. D is a mapping $S \to \mathscr{D}$, which assigns each state $s \in S$ a set of data flow guarded actions $D(s) \subseteq \mathscr{D}$ which are executed in state s.

To see an example of an EFSM, consider first the Quartz program shown in Fig. 2.15. It computes the integer approximation of the euclidean length of a `N`-dimensional vector v and does only make use of addition, subtraction, and

comparison. The correctness is not difficult to see by the given invariants in the comments:

```
next(x[i]) * next(y[i]) + next(p[i])
    = x[i] * (y[i] - 1) + (p[i] + x[i])
    = x[i] * y[i] + p[i]

next(y[0])
    = y[0] + x[0] + x[0] + 3
    = (x[0]+1)^2 + 2*x[0] + 3
    = x[0]^2 + 2*x[0] + 1 + 2*x[0] + 3
    = x[0]^2 + 4*x[0] + 4
    = (x[0]+2)^2
    = (next(x[0])+1)^2
```

The corresponding EFSM of the program shown in Fig. 2.15 is shown in Fig. 2.16 with seven states. Each state is given an index and lists the guarded actions that can

```
macro N = 2;

module VectorLength([N]int v, int !len, event !rdy) {
    [N]int p, x, y;
    // compute the squares of each v[i] in p[i] in parallel
    for(i=0..N-1) do || {
        x[i] = v[i];
        y[i] = v[i];
        p[i] = 0;
        while(y[i]>0) {
            // invariant: v[i]*v[i] = x[i]*y[i]+p[i]
            next(p[i]) = p[i] + x[i];
            next(y[i]) = y[i] - 1;
            w0: pause;
        }
    }
    // add all p[i] in p[0]
    next(p[0]) = sum(i=0..N-1) p[i];
    w1: pause;
    // now compute the square root of p[0]
    x[0] = 0;
    y[0] = 1;
    while(y[0]<=p[0]) {
        // invariant: y[0] = (x[0]+1)^2
        next(x[0]) = x[0] + 1;
        next(y[0]) = y[0] + x[0] + x[0] + 3;
        w2: pause;
    }
    // return the length of the vector v
    emit(rdy);
    len = x[0];
}
```

Fig. 2.15 The `VectorLength` Quartz program

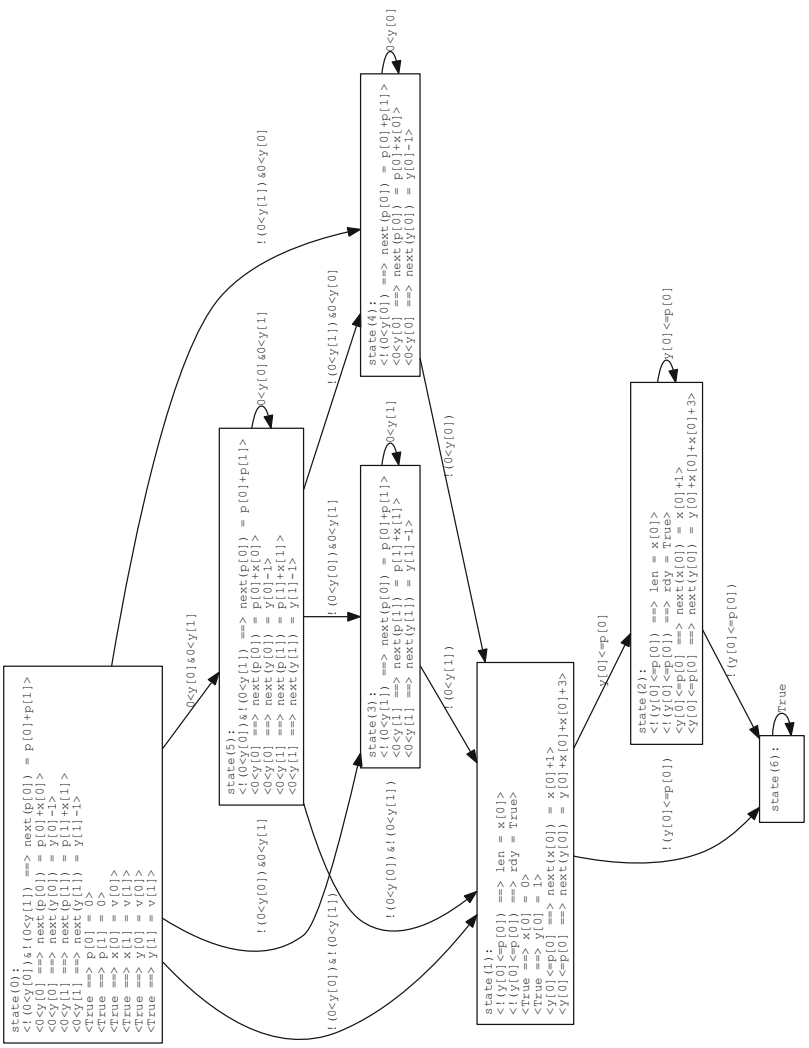


Fig. 2.16 The EFSM of `VectorLength` Quartz program for *two*-dimensional vectors

be enabled in that control flow state. State 0 contains thereby the surface of the program where the initialization of variables `x,y,p` of the for-loop and possibly the assignments in its while-loop are executed. In state 5, both while-loops for computing the squares of `v[0]` and `v[1]` are active, while in state 3 and state 4, only the one for computing the squares of `v[1]` and `v[0]` continues with the execution. Note that these states may also execute the summation `next(p[0])= p[0]+p[1]` in case the loop terminates. State 1 corresponds with label `w1` and executes the code between `w1` and `w2`, and finally state 2 executes the code from `w2` back to `w2` or leaving the loop. State 6 is a final sink state that is always added for technical reasons.

A naive way to generate the EFSM states is to take all possible 2^n states and compute the transitions from each one. However, as many states are generally not reachable, that algorithm would always need exponential time, even for programs that lead to compact EFSMs. Therefore, a better way to compute the EFSM is an abstract simulation of the program according to the operational semantics of the Quartz language.

As the control flow is explicitly enumerated, EFSMs may suffer from state-space explosion since n control flow locations may result in 2^n EFSM states. It is not only the amount of (control flow) states that poses problems, but the guarded actions for the data flow must be also replicated. However, for many practical examples, the EFSM size is still manageable, and due to the performed pre-computation, it can be optimized in many ways and can produce the fastest target code at the end. It is straightforward to generate a sequential program from an EFSM: For example, we can first define for every state a sequential code starting with a unique label and ending with a goto statement to the next code fragment of the corresponding target state.

It is important to see the difference between an EFSM and control flow graphs used in classic compiler design. While "states" of classic control/data flow graphs consist of assignments that are sequentially executed, states of the EFSM contain still guarded actions that are concurrently executed within one macro step. Moreover, transitions in the EFSM terminate a macro step of the synchronous model, so that new values of the input variables are read on the transition. Due to these differences, many transformations made in classic code optimization cannot directly be applied on EFSMs for code generation of synchronous programs.

2.6 Conclusions and Future Extensions

The synchronous model of computation can perfectly model reactive systems since its programming paradigm directly reflects the execution steps of these systems: Within a reaction step, inputs are read, and outputs are immediately computed as the reaction to these inputs. We derived the language Quartz from the classic Esterel language and modified its syntax and semantics to allow a more convenient description of hardware circuits. We also developed a formally verified compilation to synchronous guarded actions that are used as internal representation in our design

framework Averest. Using the guarded actions, various analyses are performed, in particular, the causality analysis, so that robust and deterministic system models are guaranteed. For causally correct systems, we can then generate both hardware circuits and programs, where the latter can now also be done with multiple threads. Future extensions of the language cover clocked signals that can be absent at some points of time which removes the need to synchronize generated software threads between the macro steps by completely desynchronizing the threads. Moreover, the current version of Quartz already supports hybrid systems so that one can also model a discrete system in its physical environment, e.g., for simulation or formally proving important safety properties.

References

1. Benveniste A, Bournai P, Gautier T, Le Guernic P (1985) SIGNAL: a data flow oriented language for signal processing. Research report 378, Institut National de Recherche en Informatique et en Automatique (INRIA), Rennes
2. Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages twelve years later. Proc IEEE 91(1):64–83
3. Berry G (1992) A hardware implementation of pure Esterel. Sadhana 17(1):95–130
4. Berry G (1999) The constructive semantics of pure Esterel. http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf
5. Berry G (2000) The Esterel v5 language primer. ftp://ftp.inrialpes.fr/pub/synalp/reports/esterel-primer.pdf.gz
6. Berry G, Gonthier G (1992) The Esterel synchronous programming language: design, semantics, implementation. Sci Comput Program 19(2):87–152
7. Boussinot F (1998) SugarCubes implementation of causality. Research report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis
8. Brandt J (2013) Synchronous models for embedded software. Master's thesis, Department of Computer Science, University of Kaiserslautern. Habilitation
9. Brandt J, Gemünde M, Schneider K, Shukla S, Talpin JP (2012) Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. Des Autom Embed Syst (DAEM). doi:10.1007/s10617-012-9087-9
10. Brandt J, Gemünde M, Schneider K, Shukla S, Talpin JP (2013) Embedding polychrony into synchrony. IEEE Trans Softw Eng (TSE) 39(7):917–929
11. Brandt J, Schneider K (2011) Separate translation of synchronous programs to guarded actions. Internal report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern
12. Brandt J, Schneider K, Shukla S (2010) Translating concurrent action oriented specifications to synchronous guarded actions. In: Lee J, Childers B (eds) Languages, compilers, and tools for embedded systems (LCTES). ACM, Stockholm, pp 47–56
13. Brzozowski J, Seger CJ (1995) Asynchronous circuits. Springer, New York/Berlin
14. Cassandras C, Lafortune S (2008) Introduction to discrete event systems, 2nd edn. Springer, New York
15. Chandy K, Misra J (1989) Parallel program design. Addison-Wesley, Austin
16. Closse E, Poize M, Pulou J, Sifakis J, Venter P, Weil D, Yovine S (2001) TAXYS: a tool for the development and verification of real-time embedded systems. In: Berry G, Comon H, Finkel A (eds) Computer aided verification (CAV). LNCS, vol 2102. Springer, Paris, pp 391–395
17. Closse E, Poize M, Pulou J, Venier P, Weil D (2002) SAXO-RT: interpreting Esterel semantics on a sequential execution structure. Electron Notes Theor Comput Sci (ENTCS) 65(5):80–94. Workshop on synchronous languages, applications, and programming (SLAP)

18. Dill D (1996) The Murphi verification system. In: Alur R, Henzinger T (eds) Computer aided verification (CAV). LNCS, vol 1102. Springer, New Brunswick, pp 390–393
19. Edwards S (2002) An Esterel compiler for large control-dominated systems. IEEE Trans Comput Aided Des Integr Circuits Syst (T-CAD) 21(2):169–183
20. Edwards S (2003) Making cyclic circuits acyclic. In: Design automation conference (DAC). ACM, Anaheim, pp 159–162
21. Edwards S, Kapadia V, Halas M (2004) Compiling Esterel into static discrete-event code. In: Synchronous languages, applications, and programming (SLAP), Barcelona
22. Gamatie A (2010) Designing embedded systems with the SIGNAL programming language. Springer, New York
23. Gamatié A, Gautier T, Le Guernic P, Talpin J (2007) Polychronous design of embedded real-time applications. ACM Trans Softw Eng Methodol (TOSEM) 16(2), Article 9. http://dl.acm.org/citation.cfm?id=1217298
24. Halbwachs N, Maraninchi F (1995) On the symbolic analysis of combinational loops in circuits and synchronous programs. In: Euromicro conference. IEEE Computer Society, Como
25. Harel D, Naamad A (1996) The STATEMATE semantics of statecharts. ACM Trans Softw Eng Methodol (TOSEM) 5(4):293–333
26. Howard W (1980) The formulas-as-types notion of construction. In: Seldin J, Hindley J (eds) To H.B. Curry: essays on combinatory logic, lambda-calculus and formalism. Academic, London/New York, pp 479–490
27. Järvinen H, Kurki-Suonio R (1990) The DisCo language and temporal logic of actions. Technical report 11, Tampere University of Technology, Software Systems Laboratory
28. Ju L, Huynh B, Chakraborty S, Roychoudhury A (2009) Context-sensitive timing analysis of Esterel programs. In: Design automation conference (DAC). ACM, San Francisco, pp 870–873
29. Ju L, Khoa Huynh, B., Roychoudhury A, Chakraborty S (2010) Timing analysis of Esterel programs on general purpose multiprocessors. In: Sapatnekar S (ed) Design automation conference (DAC). ACM, Anaheim, pp 48–51
30. Lamport L (1991) The temporal logic of actions. Technical report 79, Digital Equipment Cooperation
31. Li YT, Malik S (1999) Performance analysis of real-time embedded software. Kluwer, Boston/Dordrecht
32. Logothetis G, Schneider K (2003) Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In: Design, automation and test in Europe (DATE). IEEE Computer Society, Munich, pp 10196–10203
33. Malik S (1993) Analysis of cyclic combinational circuits. In: International conference on computer-aided design (ICCAD). IEEE Computer Society, Santa Clara, pp 618–625.
34. Malik S (1994) Analysis of cycle combinational circuits. IEEE Trans Comput Aided Des Integr Circuits Syst (T-CAD) 13(7):950–956
35. Poigné A, Holenderski L (1995) Boolean automata for implementing pure Esterel. Arbeitspapiere 964, GMD, Sankt Augustin
36. Potop-Butucaru D, de Simone R (2003) Optimizations for faster execution of Esterel programs. In: Formal methods and models for codesign (MEMOCODE). IEEE Computer Society, Mont Saint-Michel, pp 227–236
37. Potop-Butucaru D, Edwards S, Berry G (2007) Compiling Esterel. Springer, Boston
38. Rocheteau F, Halbwachs N (1992) Implementing reactive programs on circuits: a hardware implementation of LUSTRE. In: de Bakker J, Huizing C, de Roever WP, Rozenberg G (eds) Real-time: theory in practice. LNCS, vol 600. Springer, Mook, pp 195–208
39. Rocheteau F, Halbwachs N (1992) Pollux: a Lustre-based hardware design environment. In: Quinton P, Robert Y (eds) Algorithms and parallel VLSI architectures II. Elsevier, Gers, pp 335–346
40. Schneider K (2000) A verified hardware synthesis for Esterel. In: Rammig F (ed) Distributed and parallel embedded systems (DIPES). Kluwer, Schloß Ehringerfeld, pp 205–214

41. Schneider K (2001) Embedding imperative synchronous languages in interactive theorem provers. In: Application of concurrency to system design (ACSD). IEEE Computer Society, Newcastle Upon Tyne, pp 143–154
42. Schneider K (2002) Proving the equivalence of microstep and macrostep semantics. In: Carreño V, Muñoz C, Tahar S (eds) Theorem proving in higher order logics (TPHOL). LNCS, vol 2410. Springer, Hampton, pp 314–331
43. Schneider K (2009) The synchronous programming language Quartz. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern
44. Schneider K, Brandt J (2008) Performing causality analysis by bounded model checking. In: Application of concurrency to system design (ACSD). IEEE Computer Society, Xi'an, pp 78–87
45. Schneider K, Brandt J, Schüle T (2004) Causality analysis of synchronous programs with delayed actions. In: Compilers, architecture, and synthesis for embedded systems (CASES). ACM, Washington, pp 179–189
46. Schneider K, Brandt J, Schüle T (2004) A verified compiler for synchronous programs with local declarations (proceedings version). In: Synchronous languages, applications, and programming (SLAP), Barcelona
47. Schneider K, Brandt J, Schüle T (2006) A verified compiler for synchronous programs with local declarations. Electron Notes Theor Comput Sci (ENTCS) 153(4):71–97
48. Schneider K, Brandt J, Schüle T, Türk T (2005) maximal causality analysis. In: Desel J, Watanabe Y (eds) Application of concurrency to system design (ACSD). IEEE Computer Society, Saint-Malo, pp 106–115
49. Schneider K, Wenz M (2001) A new method for compiling schizophrenic synchronous programs. In: Compilers, architecture, and synthesis for embedded systems (CASES). ACM, Atlanta, pp 49–58
50. Schüle T, Schneider K (2004) Abstraction of assembler programs for symbolic worst case execution time analysis. In: Malik S, Fix L, Kahng A (eds) Design automation conference (DAC). ACM, San Diego, pp 107–112
51. Shiple T (1996) Formal analysis of synchronous circuits. PhD thesis, University of California, Berkeley
52. Shiple T, Berry G, Touati H (1996) Constructive analysis of cyclic circuits. In: European design automation conference (EDAC). IEEE Computer Society, Paris, pp 328–333
53. Shiple T, Singhal V, Brayton R, Sangiovanni-Vincentelli A (1996) Analysis of combinational cycles in sequential circuits. In: International symposium on circuits and systems (ISCAS), Atlanta, pp 592–595
54. Tardieu O, de Simone R (2004) Curing schizophrenia by program rewriting in Esterel. In: Formal methods and models for codesign (MEMOCODE). IEEE Computer Society, San Diego, pp 39–48

SysteMoC: A Data-Flow Programming Language for Codesign

3

Joachim Falk, Christian Haubelt, Jürgen Teich, and Christian Zebelein

Abstract

Computations in hardware/software systems are inherently performed concurrently. Hence, modeling hardware/software systems requires notions of concurrency. Data-flow models have been and are still successfully applied in the modeling of hardware/software systems. In this chapter, we motivate and introduce the usage of data-flow models. Moreover, we discuss the expressiveness and analyzability of different data-flow Models of Computation (MoCs). Subsequently, we present SysteMoC, an approach supporting many data-flow MoCs based on the system description language SystemC. Besides specifying data-flow models, SystemMoC also permits the automatic classification of each different part of an application modeled in SysteMoC into a least expressive but most analyzable MoC. This classification is the key to further optimization in later design stages of hardware/software systems such as exploration of design alternatives as well as automatic code generation and hardware synthesis. Such optimization and refinement steps are employed as part of the SYSTEMCODESIGNER design flow that uses SysteMoC as its input language.

J. Falk (✉) • J. Teich
Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany
e-mail: joachim.falk@fau.de; juergen.teich@fau.de

C. Haubelt
Department of Computer Science and Electrical Engineering, Institute of Applied Microelectronics and Computer Engineering, University of Rostock, Rostock, Germany
e-mail: christian.haubelt@uni-rostock.de

C. Zebelein
Valeo Siemens eAutomotive Germany GmbH, Erlangen, Germany
e-mail: christian.zebelein.jv@valeo-siemens.com

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_4

Acronyms	
BDF	Boolean Data Flow
CIC	Common Intermediate Code
CPU	Central Processing Unit
CSDF	Cyclo-Static Data Flow
DDF	Dennis Data Flow
DFG	Data-Flow Graph
DSE	Design Space Exploration
FIFO	First-In First-Out
FSM	Finite-State Machine
FunState	Functions Driven by State Machines
HSCD	Hardware/Software Codesign
HSDF	Homogeneous (Synchronous) Data Flow
KPN	Kahn Process Network
MoC	Model of Computation
NDF	Non-Determinate Data Flow
SDF	Synchronous Data Flow
SysteMoC	SystemC Models of Computation

Contents

3.1 Introduction

Due to the rising capabilities of embedded systems, their complexity has also increased tremendously. As a consequence, embedded systems are no longer implemented on a single computational resource, but in the form of a complex hardware/software system consisting of multiple connected heterogeneous resources including processor cores, hardware accelerators, and complex communication infrastructure to connect all these components. Hence, languages used

for implementing and modeling applications to be mapped to such embedded systems need the ability to reflect and exploit the parallelism inherent in such target architectures.

> Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism.
>
> — From "The Problem with Threads," by Edward A. Lee [27]

The above observation causes problems for the traditional implementation of embedded systems via sequential programming languages, as these languages typically handle concurrency only by using threads. *Data flow*, in contrast, is a modeling paradigm well-suited for the modeling of concurrent systems by concurrent *actors* that perform computation depending on the availability of *tokens* carrying data transmitted between them via First-In First-Out (FIFO) channels. Thus, data-flow models are particularly useful for modeling streaming applications as commonly found in the multimedia or networking domain and, hence, are a natural fit for modeling embedded systems, which should be implemented as codesigned hardware/software systems.

Over the last decades, many data-flow Models of Computation (MoCs) have been developed. They are usually classified according to their *expressiveness*, i.e., which kind of applications can be modeled by using a given data-flow MoC. It can be observed that *analyzability* of data-flow MoCs is inversely related to their expressiveness, i.e., there are problems which are decidable for less expressive data-flow MoCs, but are not decidable for more expressive ones (see Fig. 3.1). As analyzability of properties such as required bandwidth of channels, deadlock freedom, or schedulability issues is very important in early design phases of an embedded system, data-flow MoCs with a high analyzability are usually desirable. For example, scheduling of actors on computational resources at compile time (*static scheduling*) is usually preferred over scheduling at run time (*dynamic scheduling*) in order to reduce the overhead incurred by the scheduling strategy. However, static schedules can only be computed for data-flow MoCs with limited expressiveness, as discussed below.

In Sect. 3.2, we give a survey and classify different data-flow models starting with *static* data-flow models and subsequently increasing the expressiveness of the models up to Non-Determinate Data Flow (NDF) [26]. Based on these discussions, we will introduce the *SystemC Models of Computation (SysteMoC) modeling language* [11,12] by example in Sect. 3.3. This language has strong formal underpinnings in data-flow modeling, but with the distinction that the expressiveness of the data-flow model used by an actor is not chosen in advance but determined from the implementation of the actor [12, 45]. Later on, in Sect. 3.4, we will provide a definition of the formal semantics of the language. To support modeling of even very complex real-world applications, the SysteMoC language realizes the NDF model. Nonetheless, to enable some SysteMoC applications to reap the benefits of analyzability of less expressive MoCs, the SysteMoC language—in

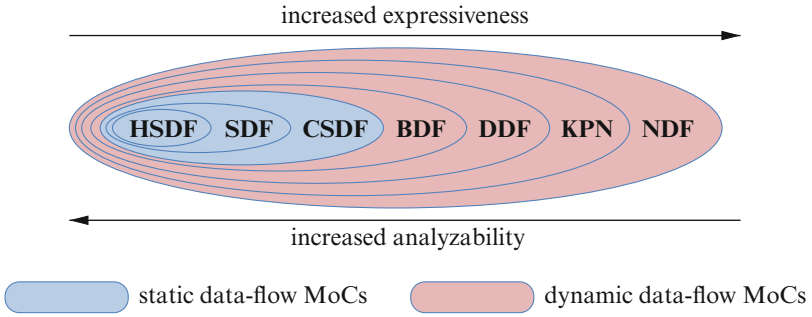


Fig. 3.1 Depicted above is a hierarchy [40] of various data-flow MoCs. The hierarchy is partitioned into static (*light blue*) data-flow MoCs and dynamic (*light red*) data-flow MoCs. A detailed explanation of the three static data-flow MoCs Homogeneous (Synchronous) Data Flow (HSDF), Synchronous Data Flow (SDF), and Cyclo-Static Data Flow (CSDF) that are well known from literature will be given in Sect. 3.2.2. Moreover, the dynamic data-flow MoCs Boolean Data Flow (BDF), Dennis Data Flow (DDF), Kahn Process Networks (KPNs), and Non-Determinate Data Flow (NDF) will be discussed in Sect. 3.2.3

contrast to general design languages such as SystemC—enforces a distinction between communication and computation of an actor. In Sect. 3.5, it will be shown how this distinction between communication and computation can be exploited in order to classify a SysteMoC actor into one of the static data-flow models (light blue in Fig. 3.1) in the hierarchy of data-flow model expressiveness. Hence, if the high expressiveness of SysteMoC is not used by a SysteMoC actor, then analysis techniques may detect this and classify the actor into a data-flow model of lower expressiveness but higher analyzability. The classification provides only a sufficient criterion if a general SysteMoC actor conforms to one of the static data-flow models. This limitation stems from the fact that the problem in general is undecidable. The chapter concludes with an overview of the SystemCoDesigner, a codesign framework based on SysteMoC as its design language. Here, we will give examples how SysteMoC may not only be used to integrate with Design Space Exploration (DSE) (More details on the DSE part within SystemCoDesigner can be found in ▸ Chap. 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration".), but also subsequent hardware/software code generation in the refinement to its final implementation.

3.2 Overview of Basic Data-Flow Models

Data flow is a modeling paradigm well-suited for the modeling of concurrent systems by concurrently executing actors. Thus, data flow is a natural fit for the modeling of embedded hardware/software systems where the interaction and execution (control) of its functions is ruled by the availability of data. In the case of control-dominated systems, the *synchronous approach* presented in ▸ Chap. 2, "Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems" is more convenient for their modeling. In the following, a general

introduction to data flow is given in Sect. 3.2.1. Moreover, the trade-off between analyzability and expressiveness of these MoCs will be discussed. Data-flow models can be classified into *static* data-flow models, known for their high analyzability but low expressiveness, and *dynamic* data-flow models, known for their high expressiveness but low analyzability. Next, three important static data-flow models HSDF [6], SDF [28], and CSDF [3] known from literature will be recapitulated in Sect. 3.2.2. Finally, in Sect. 3.2.3, dynamic data-flow models such as KPNs [24] as well as usual realizations of them in the form of DDF [7] and BDF [5] are discussed.

3.2.1 Data Flow

A Data-Flow Graph (DFG) [24] is a graph consisting of vertices called *actors* and directed edges called *channels*. Whereas actors are used to model functionality, thus computations to be executed, channels represent data communication and storage requiring memory for their implementation. If not otherwise stated, channel memory is conceptually considered to be unbounded, thus representing a possibly infinite amount of storage. Moreover, the computation of an actor is usually [7] separated into distinct steps. These steps are called *actor firings*. An actor firing is an atomic computation step that consumes a number of data items called *tokens* from each incoming channel and produces a number of tokens on each outgoing channel. More formally, a DFG is defined as follows:

Definition 1 (Data-Flow Graph). A DFG is a directed graph $g = (A, C)$, where the set of vertices A represents the set of *actors* and the set of edges $C \subseteq A \times A$ represents the set of *channels*. Additionally, a *delay function* $\mathbf{delay} : C \rightarrow \mathscr{V}^*$ is given. (The "*"-operator is used to denote Kleene closure of a value set. It generates the set of all possible finite and infinite sequences of values from the value set, that is $X^* = \cup n \in \mathbb{N}_0 : X^n$. $\mathbb{N}_0$ denotes the set of non-negative integers, that is $\{0, 1, 2, \ldots\}$.) It assigns to each channel $(a_{\text{src}}, a_{\text{snk}}) = c \in C$ a (possibly empty) sequence of initial tokens. (In some data-flow models that abstract from token values, the delay function may only return a non-negative integer that denotes the number of initial tokens on the channel. In such a context, the number of initial tokens may also be called the *delay* of a channel.) The set $\mathscr{V}$ is the set of data values which can be carried by a token. Finally, a channel capacity can be stated via the channel capacity function $\mathbf{size} : C \rightarrow \mathbb{N}_0$ that denotes the maximal number of tokens a channel can store.

An example of a very simple DFG according to Definition 1 is depicted in Fig. 3.2. It consists of two actors a_1 and a_2 which are connected by a channel c_1. A channel has, if not otherwise stated, an *infinite channel capacity*, i.e., it can store an infinite number of tokens that are in transit between the two actors connected by the channel. For notational convenience, the **src** and **snk** functions are used to refer, respectively, to the source actor, e.g., $\mathbf{src}(c_1) = a_1$, and the sink actor, e.g., $\mathbf{snk}(c_1) = a_2$, of a channel.

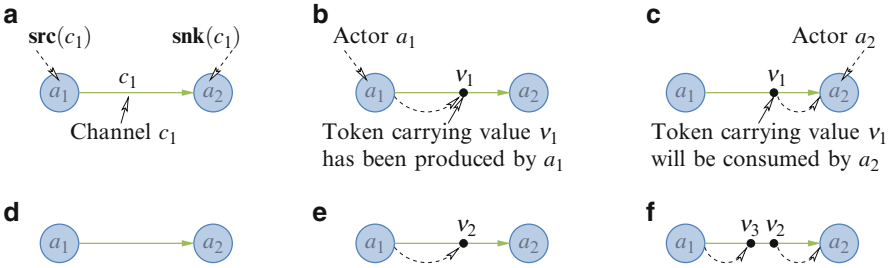


Fig. 3.2 A DFG consisting of a single channel communicating data produced by actor a_1 and consumed by actor a_2. (**a**) Initial state of the DFG g. (**b**) Token production by actor a_1. (**c**) Token consumption by actor a_2. (**d**) Initial state is again reached after consumption of the token. (**e**) After production of a second token by actor a_1. (**f**) Production of a third token by a_1 while actor a_2 is consuming the second token

The channel c_1 enables a transmission of data values $v_1, v_2, v_3, \ldots$ from a_1 to a_2. Each data value is carried by a *token*. For data-flow models, a *token* represents the atomic unit of data production and consumption. Tokens are generally queued (as exemplified in Fig. 3.2b, e, f) and de-queued (Fig. 3.2c, f) on a channel in FIFO order. Between Fig. 3.2a, b, an *actor firing* of actor a_1 has occurred, producing a token with value v_1. Next, in Fig. 3.2c, the first actor firing of actor a_2 consumes the token and its contained data value v_1. The state of a DFG is given by the number and values of tokens on each channel as well as, possibly, the internal states of all actors. For analysis purposes of static data-flow models, the values of these tokens as well as the internal state of all actors may be ignored. Hence, a state equivalent to the initial state is again reached in Fig. 3.2d. The significance of this observation will be discussed in more detail in Sect. 3.2.2.1.

In Fig. 3.2d–f, another two tokens (v_2 and v_3) are produced by actor a_1 and the second token (v_2) is in the process of being consumed by actor a_2. A resulting next state of the DFG after the second firing of actor a_2 is not depicted in Fig. 3.2, but consists of the DFG where only the token with value v_3 remains on the channel.

An actor is *fireable* (also called *enabled*) if and only if all the tokens the next actor firing will consume are present on the input channels of the actor, e.g., the actor a_2 is enabled in Fig. 3.2b, c, e, f as a token, which is the only token that will be consumed by a firing of a_2, is present on the channel.

The literature on data-flow MoCs is very broad. Indeed, decades of research [1, 3, 5–7, 11–14, 16–19, 21, 22, 24, 28, 29, 31, 33, 34, 41, 45] into its applications have led to a multitude of different data-flow models. All of them make different trade-offs between their expressiveness and their analyzability, e.g., with respect to *deadlock freedom*, the ability to be executed in *bounded memory*, or the possibility to be *scheduled at compile time*. In the following, the most important data-flow MoCs are briefly reviewed, starting with those data-flow models with the least expressiveness.

3.2.2 Static Data Flow

Of primary interest for the expressiveness of a data-flow model are *production* and *consumption rates*. The *production rate* ($\mathbf{prod}(c)$) of an actor firing to a connected channel c is the number of tokens which are produced by that actor on the channel while firing. Consider Fig. 3.2b as an example where the first firing of actor a_1 produces one token onto channel c_1. Therefore, the production rate of the first firing of actor a_1 to channel c_1 is one. The *consumption rate* ($\mathbf{cons}(c)$) is defined analogously, e.g., Fig. 3.2c depicts a situation where the first firing of actor a_2 consumes one token from channel c_1. Thus, the consumption rate of the first firing of actor a_2 from channel c_1 is one. That is, the *consumption rate* of an actor firing from a connected channel is the number of tokens which are consumed by that actor from the channel while firing.

Now, an actor is called a *static data-flow actor* if its production and consumption rates are (1) *not dependent on the values of the tokens which are consumed by the actor*, (2) *not dependent on the points in time at which tokens arrive on the input channels or free places become available at the output channels*, and (3) *not dependent on some random process*. Thus, the communication behavior of a static actor can be fully predicted at compile time. The *consumption* and *production rates* of an actor are even further constrained in well-known static data-flow models, which are presented in the following.

3.2.2.1 Homogeneous Data Flow

The simplest data-flow model is Homogeneous (Synchronous) Data Flow (HSDF). (Note that the term *synchronous* in the name of two data-flow MoCs introduced in this chapter was coined in the original paper [29], but is, unfortunately, totally independent from the semantics of the term as used for *synchronous languages* that were introduced in ▸ Chap. 2, "Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems".) Data-flow graphs corresponding to this data-flow model are also known as *marked graphs* [6] in literature. The *communication behavior* of HSDF actors is constrained in such a way that each actor firing must produce and consume exactly one token on, respectively, each outgoing and incoming channel, i.e., $\forall c \in C : \mathbf{cons}(c) = \mathbf{prod}(c) = 1$. Due to the low modeling power of the HSDF model, it is also highly analyzable; e.g., if each actor in an HSDF has fired exactly once, then the graph will be back to its initial state. If we assume that the DFG depicted in Fig. 3.2 is an HSDF graph, then firing actors a_1 and a_2 both once will transmit one token (ν_1) over the channel and lead back to the initial state (shown in Fig. 3.2a–d) where no token is present on the channel. These two actor firings $\langle a_1, a_2 \rangle$ realize a so-called *iteration* of the graph. It is proven in [6] that an HSDF graph has such an iteration if and only if each directed cycle of the graph contains at least one *initial token*. Briefly, a deadlock results in case there exists a cycle without any initial tokens in the graph because each actor that is part of this cycle can never fire as it awaits the production of a token by its predecessor actor in the cycle.

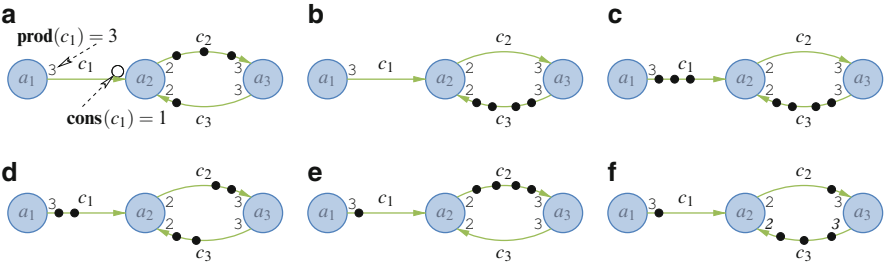


Fig. 3.3 Example of a Synchronous Data Flow graph and a sequence of actor firings realizing the iteration of the graph. (**a**) Initial state of the SDF graph. (**b**) After firing of actor a_3. (**c**) After firing of actor a_1. (**d**) After firing of actor a_2. (**e**) After firing of actor a_2. (**f**) After firing of actor a_3 (The next firing of actor a_2 will lead back to the initial state)

3.2.2.2 Synchronous Data Flow

In the Synchronous Data Flow (SDF) [29] MoC, the communication behavior of the actors is constrained to have consumption and production rates being *constant for all firings of an actor*. Moreover, consumption and production rates for all connected channels are assumed to be arbitrary positive integer constants. Therefore, in SDF, the consumption and production rates can be expressed by the *consumption rate function* $\mathbf{cons} : C \to \mathbb{N}$ and the *production rate function* $\mathbf{prod} : C \to \mathbb{N}$ that specify for each channel $c \in C$, respectively, the number of consumed tokens from the channel by an actor firing of the actor $\mathbf{snk}(c)$ and the number of produced tokens onto the channel by an actor firing of the actor $\mathbf{src}(c)$. (The symbol $\mathbb{N}$ is used to denote the set of natural numbers, that is the set $\{1, 2, 3, \ldots\}$.)

In visual representations of SDF graphs, as usual, the consumption and production rates are annotated at the beginnings and endings of the channel edges. An example of an SDF graph is depicted in Fig. 3.3a. Consumption and production rates of one, e.g., $\mathbf{cons}(c_1) = 1$, are traditionally not annotated to reduce clutter.

The question arises which conditions are *necessary and sufficient* for the existence of an iteration of a SDF graph. It is proven in [28] that such an iteration can be determined by *balance equations*. Each balance equation corresponds to one channel in the SDF graph. The balance equation for a channel c is given as: $\eta_{\mathbf{src}(c)} \cdot \mathbf{prod}(c) = \eta_{\mathbf{snk}(c)} \cdot \mathbf{cons}(c)$, where the variable $\eta_{\mathbf{src}(c)}$ denotes the number of actor firings of the actor producing tokens onto channel c while $\eta_{\mathbf{snk}(c)}$ denotes the number of actor firings of the actor consuming tokens from channel c. Given $\mathbf{prod}(c)$ and $\mathbf{cons}(c)$, the left and right sides of the equation denote the number of tokens that have been produced and consumed by the $\eta_{\mathbf{src}(c)}$ source actor and $\eta_{\mathbf{snk}(c)}$ sink actor firings, respectively. For the graph depicted in Fig. 3.3a, the balance equations corresponding to the three channels c_1 to c_3 of the graphs are as follows:

$$\eta_{a_1} \cdot 3 = \eta_{a_2} \cdot 1 \quad \eta_{a_2} \cdot 2 = \eta_{a_3} \cdot 3 \quad \eta_{a_3} \cdot 3 = \eta_{a_2} \cdot 2 \tag{3.1}$$

For an iteration, both sides of the equation must balance, otherwise an iteration would not bring the graph into the same state as it has started from. Hence, a solution besides the trivial zero solution is a *necessary* condition [28] for the existence of an iteration. If there exists such a solution for a static DFG, then this graph is called *consistent*.

In Equation (3.1), the number of firings for the actors a_1 to a_3 are given by η_{a_1} to η_{a_3}, respectively. Using the convention established in [2], the *minimum positive integer solution* to the set of balance equations, e.g., $\boldsymbol{\eta}^{\text{rep}} = (\eta_{a_1}, \eta_{a_2}, \eta_{a_3}) = (1, 3, 2)$, is called the *repetition vector* $\boldsymbol{\eta}^{\text{rep}}$ of an SDF graph. The length of this iteration is determined by summing all the entries of the repetition vector, e.g., for the SDF graph in Fig. 3.3a, the iteration can be realized by a sequence $\langle a_3, a_1, a_2, a_2, a_3, a_2 \rangle$ of $\eta_{a_1} + \eta_{a_2} + \eta_{a_3} = 6$ actor firings as shown in Fig. 3.3a–f.

However, the existence of a repetition vector does not guarantee that a sequence of actor firings can be found that realizes the iteration. To exemplify, the four initial tokens of the SDF graph depicted in Fig. 3.3a are removed. This does not change the calculation or existence of the *repetition vector* of the SDF graph. However, without any initial tokens, neither actor a_2 nor actor a_3 can ever be fired. In general, a *necessary and sufficient criterion of the existence of the iteration* is to test whether a computed repetition vector may also execute as an iterative deadlock-free schedule by firing each fireable actor as many times as implied by the repetition vector until the iteration is finished or a deadlock has occurred. Note that in the general case, the length of the iteration may be exponential in the size of the SDF graph. Hence, in contrast to HSDF models, where the question of the existence of an iteration can be answered in polynomial time [20], the problem is only solvable in exponential time for SDF graphs [32].

3.2.2.3 Cyclo-Static Data Flow

An extension of the SDF model is Cyclo-Static Data Flow (CSDF). In the CSDF [3] MoC, the communication behavior of an actor is extended to allow for cyclically varying consumption and production rates between actor firings. The length of this cycle is known as the number τ of *phases* of a CSDF actor. An actor firing of a CSDF actor is also known as a CSDF *phase*. To accommodate the cyclically varying consumption and production rates, the consumption and production rate functions have to be extended to return vectors of length τ (the number of phases of the CSDF actor consuming or producing tokens), i.e., the functions $\mathbf{cons} : C \to \mathbb{N}_0^\tau$ and $\mathbf{prod} : C \to \mathbb{N}_0^\tau$ specify for each channel $c \in C$ a vector $(n_0, n_1, \ldots, n_{\tau-1})$ where each vector entry corresponds to the consumption or production rate of the CSDF actor in the corresponding phase.

The question whether there exists an iteration can again be answered by solving the appropriate balance equations—answering the question of the consistency of the CSDF graph—and if the graph is consistent by testing whether the computed repetition vector may also execute as an iterative deadlock-free schedule. For the purpose of calculating the repetition vector, all CSDF actors can be replaced by SDF actors with consumption and production rates derived by summing all rates in the corresponding vectors of consumption and production rates of the CSDF actors.

Hence, the balance equation for a channel c from actor a_{src} to actor a_{snk} with the respective production and consumption rates $\mathbf{prod}(c) = (n_0, n_1, \ldots n_{\tau_{\text{src}}-1})$ and $\mathbf{cons}(c) = (m_0, m_1, \ldots m_{\tau_{\text{snk}}-1})$ is as follows:

$$\frac{\eta_{a_{\text{src}}}}{\tau_{\text{src}}} \cdot (n_0 + n_1 + \ldots + n_{\tau_{\text{src}}-1}) = \frac{\eta_{a_{\text{snk}}}}{\tau_{\text{snk}}} \cdot (m_0 + m_1 + \ldots + m_{\tau_{\text{snk}}-1})$$

As is the case for SDF models, the smallest positive integer solution of the *balance equations* uniquely determines the (minimal) repetition vector of the CSDF graph. Finally, the existence of a valid iteration corresponding to the repetition vector needs to be verified as—like in SDF—the existence of the repetition vector is only a necessary but not sufficient criterion for the existence of the iteration.

3.2.3 Dynamic Data Flow

In contrast to *static data-flow models*, where the consumption and production rates cannot be influenced by the values of the consumed tokens, dynamic data-flow actors can vary their consumption and production rates depending on the *history of the consumed tokens* and also *on the tokens to be consumed*. Dynamic data flow is the first introduced data-flow model where consumption and production rates depend on the data values being consumed and produced.

3.2.3.1 Boolean Data Flow

Boolean Data Flow (BDF) [5] can be seen as an extension of the class of static data-flow models by introducing two dynamic actor types, the *switch* actor and the *select* actor. In the following, the notion of ports will be used interchangeably with the channels connected to these ports. Hence, expressions like $\mathbf{cons}(i)$ and $\mathbf{prod}(o)$ are used to refer to the consumption rate and production rate on the channel connected to the respective input or output port. This enables us to depict an actor and show its implementation in isolation, e.g., as has been used in Fig. 3.4.

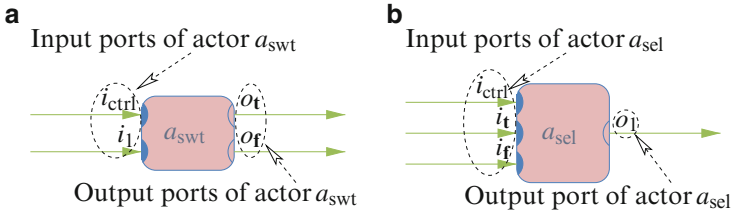


Fig. 3.4 Shown here are the BDF switch and select actors with their corresponding input and output ports. The color scheme chosen to distinguish static and dynamic actors has been selected according to the colors marking static and dynamic MoCs in Fig. 3.1. (**a**) Depiction of the switch actor a_{swt} in isolation. (**b**) Depiction of the select actor a_{sel} in isolation

Moreover, let **t** (true) and **f** (false) denote the Boolean truth values. The *switch* actor, depending on the truth value of a control token (see Fig. 3.4a) from its control input port i_{ctrl}, forwards a token from its input port i_1 to either its true ($o_{\mathbf{t}}$) or its false ($o_{\mathbf{f}}$) output port. The *select* actor acts in the opposite way, i.e., depending on the truth value of a control token (see Fig. 3.4b) from its control input port i_{ctrl}, it forwards a token from either its true ($i_{\mathbf{t}}$) or its false ($i_{\mathbf{f}}$) input port to its output port o_1.

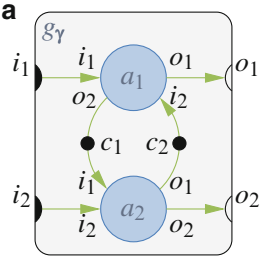
The usage of these two dynamic actor types together with the arithmetic primitives enable the construction of arbitrary control flow structures. The channels (of infinite capacity) can be used to represent an infinite memory. Simple arithmetic operations are supported by BDF via its ability to model static actors, e.g., like an SDF actor implementing an addition. Together with the control flow logic, a Turing machine can be implemented by the BDF model [5]. Hence, the existence of iterations or the problem of execution in bounded memory is in general already undecidable for BDF graphs.

3.2.3.2 Dennis Data Flow

Dennis Data Flow (DDF) [7] is an extension of BDF by allowing all dynamic actors that are realizable by using *(blocking) read* and *(blocking) write* communication primitives. Code inside DDF actors is assumed to be executed sequentially, e.g., as seen in Fig. 3.5b, c. A blocking read or write primitive is used to receive (see Lines 3 to 4) or transmit (see Lines 5 to 6) one data value on an input or output channel, respectively. Once a blocking read or write primitive is invoked, the execution of the actor will block until the data value has been successfully received or transmitted. Hence, at most one blocking read or write primitive can be active at one point in time.

All MoCs more general than BDF according to Fig. 3.1 are Turing complete. However, one aspect of difference is the modeling power of a single actor. An

a



A data-flow subgraph consisting of two HSDF actors

b

```
1: procedure a1
2:     while t do
3:         v1 ← read(a1.i1)
4:         v2 ← read(a1.i2)
5:         write(a1.o1,v1)
6:         write(a1.o2,v2)
7:     end while
8: end procedure
```

Implementation of the HSDF actor a_1 via the read and write operations provided by DDF

c

```
1: procedure a2
2:     while t do
3:         v1 ← read(a2.i1)
4:         v2 ← read(a2.i2)
5:         write(a2.o1,v1)
6:         write(a2.o2,v2)
7:     end while
8: end procedure
```

Implementation of the HSDF actor a_2 via the read and write operations provided by DDF

Fig. 3.5 Depicted above (see (**a**)) is the subgraph g_γ that can be connected with input and output ports i_1, i_2, o_1, and o_2 to an unspecified environment. The subgraph consists of two DDF actors a_1 (see (**b**)) and a_2 (see (**c**)) that both implement a communication behavior that corresponds to the HSDF MoC

important question here is whether it is possible to take an arbitrary connected subgraph of a DFG, e.g., the one shown in Fig. 3.5a, and represent it as an actor in the data-flow model. This property is called *compositionality* of the MoC. If a data-flow model is compositional, then the highly desirable operation of abstraction becomes seamlessly possible. An abstraction operation can be performed by hiding the implementation complexity of an arbitrary connected subgraph of the model behind the interface of an actor. If the data-flow model is compositional, then this *composite actor*, which represents the subgraph, can be handled like any other actor in the system. Otherwise, the composite actor, e.g., the actor a_γ that represents the subgraph g_γ, is always an exception and needs special treatment.

To exemplify, we consider the question of the least expressive MoC required to express the composite actor a_γ for the subgraph g_γ. All HSDF actors can also be represented by DDF actors, e.g., as shown in Fig. 3.5b, c for the actors a_1 and a_2 of the subgraph g_γ. In contrast, due to the constraint that DDF actors can only use (blocking) read to access data from their input ports, there is no composite actor a_γ that corresponds to the DDF MoC. Here, the problematic situation is that the composite actor a_γ has to first produce a token on either output port o_1 or output port o_2 depending on whether a token arrives first at either input port i_1 or input port i_2, respectively. However, due to the (blocking) read semantics of DDF, there is no possibility to detect on which input port a token arrives first. Once an input port has been chosen for a read access, the DDF actor cannot process any tokens from any other input port until a token has been successfully read.

Moreover, as the MoCs HSDF, SDF, CSDF, and BDF are all less expressive than DDF, there is also no composite actor a_γ that corresponds to one of these less expressive MoCs. Hence, as the actors a_1 and a_2 are of the least expressive MoC HSDF and the composite actor cannot be expressed via the DDF MoC, all MoCs up to and including DDF are non-compositional. In contrast, as will be shown in the next section, the composite actor a_γ can be expressed as the Kahn function κ_{a_γ}.

One could argue that the expressiveness of a DDF actor is not a proper superset of the expressiveness of an HSDF actor due to the issue of atomicity in the consumption and production of tokens that is implied by the notion of firing of an HSDF actor. In contrast, read or write communication primitives used by DDF actors consume or produce tokens in isolation, thus allowing the firing of other actors to interrupt a sequence of read or write communication primitives in the DDF actor. However, if we consider DFGs containing only actors of the expressiveness KPN and below, then the issue of atomicity is not a relevant criterion for the functionality of a DFG due to the *sequence determinate* [30] nature of such DFGs. Briefly, if a DFG is sequence determinate, then the behavior of the DFG is independent from the sequence of actor firings that are taken to schedule the graph. Thus, if a sequence of read or write communication primitives is interrupted by other actor firings or not does not influence the functionality of the DFG. In conclusion, the issue of atomicity is not relevant for the proof of non-compositionality of MoCs DDF and below.

If atomicity should also be considered in the hierarchy of data-flow model expressiveness, then the semantics of DDF must be extended to allow grouping of

sequences of read or write communication primitives to be executed atomically or not at all. Unsurprisingly, such an extension of the semantics of DDF still does not allow a DDF actor to model the data-flow subgraph g_γ from Fig. 3.5a.

3.2.3.3 Kahn Process Networks

Kahn Process Networks (KPNs) are one of the oldest data-flow MoCs. The original paper [24] of Kahn used a *denotational semantics* to describe the behavior of a KPN actor. In this denotational semantics, an actor a is described by a Kahn function κ_a. To exemplify, the two (identical) HSDF actors a_1 and a_2 in Fig. 3.5a are represented by the two (identical) Kahn functions κ_{a_1} and κ_{a_2} given in Equation (3.2).

$$\kappa_{a_1}(s_x, s_y) = \kappa_{a_2}(s_x, s_y) = \begin{cases} (\langle\rangle, \langle\rangle) & \text{if } \#s_x = 0 \vee \#s_y = 0 \\ (\langle \mathbf{hd}(s_x)\rangle, \langle \mathbf{hd}(s_y)\rangle) \overrightarrow{\frown} \kappa_{a_1}(\mathbf{tl}(s_x), \mathbf{tl}(s_y)) & \text{otherwise} \end{cases} \tag{3.2}$$

Here, a so-called *signal* $s \in S \equiv \mathscr{V}^*$ is (a possibly infinite) sequence of values carried by the tokens being transported over a channel, e.g., $\langle 5, 8, 7\rangle$ for a sequence of three values. *The length of a signal*, i.e., the number of values contained in it, will be denoted by the $\#s$ notation, e.g., $\#\langle 5, 8, 7\rangle = 3$. The $\mathbf{hd}(s)$ and the $\mathbf{tl}(s)$ notations are used to access the *head* of a sequence and, respectively, the *tail* of a sequence, i.e., the sequence without its head.

Considering Equation (3.2), we see that the Kahn function returns a tuple of empty sequences $(\langle\rangle, \langle\rangle)$ if at least one of the input signals s_x or s_y is empty, i.e., their length is zero. Otherwise, the Kahn function is called recursively with the tails of both input sequences, i.e., $\kappa_{a_1}(\mathbf{tl}(s_x), \mathbf{tl}(s_y))$, and the result of this computation is concatenated to the tuple of single value sequences containing the head of both input sequences, i.e., $(\langle \mathbf{hd}(s_x)\rangle, \langle \mathbf{hd}(s_y)\rangle)$. To demonstrate, we perform the following Kahn function application:

$$\begin{aligned} \kappa_{a_1}(\langle 5, 8, 7\rangle, \langle 9, 8\rangle) &= (\langle \mathbf{hd}(\langle 5, 8, 7\rangle)\rangle, \langle \mathbf{hd}(\langle 9, 8\rangle)\rangle) \overrightarrow{\frown} \kappa_{a_1}(\mathbf{tl}(\langle 5, 8, 7\rangle), \mathbf{tl}(\langle 9, 8\rangle)) \\ &= (\langle 5\rangle, \langle 9\rangle) \overrightarrow{\frown} \kappa_{a_1}(\langle 8, 7\rangle, \langle 8\rangle) = (\langle 5\rangle, \langle 9\rangle) \overrightarrow{\frown} (\langle 8\rangle, \langle 8\rangle) \overrightarrow{\frown} \kappa_{a_1}(\langle 7\rangle, \langle\rangle) \\ &= (\langle 5\rangle, \langle 9\rangle) \overrightarrow{\frown} (\langle 8\rangle, \langle 8\rangle) \overrightarrow{\frown} (\langle\rangle, \langle\rangle) = (\langle 5\rangle \frown \langle 8\rangle \frown \langle\rangle, \langle 9\rangle \frown \langle 8\rangle \frown \langle\rangle) \\ &= (\langle 5, 8\rangle, \langle 9, 8\rangle) \end{aligned}$$

Here, the "$\frown$"-operator is used to concatenate two sequences, and the "$\overrightarrow{\frown}$"-operator is applied to tuples of signals by pointwise extension of the "$\frown$"-operator.

In general, a Kahn function $\kappa_a : S^n \to S^m$ transforms n value sequences on its n input ports to m value sequences on its m output ports. All Kahn functions are required [30] to be Scott-continuous. In practice, this means that appending values to the input sequences of the Kahn function can only result in appending values to the resulting output sequences, i.e., if $\kappa_a(s_{i_1}, s_{i_2}, \ldots s_{i_n}) = (s_{o_1}, s_{o_2}, \ldots s_{o_m})$, then $\kappa_a(s_{i_1} \frown s'_{i_1}, s_{i_2} \frown s'_{i_2}, \ldots s_{i_n} \frown s'_{i_n}) = (s_{o_1} \frown s'_{o_1}, s_{o_2} \frown s'_{o_2}, \ldots s_{o_m} \frown s'_{o_m})$.

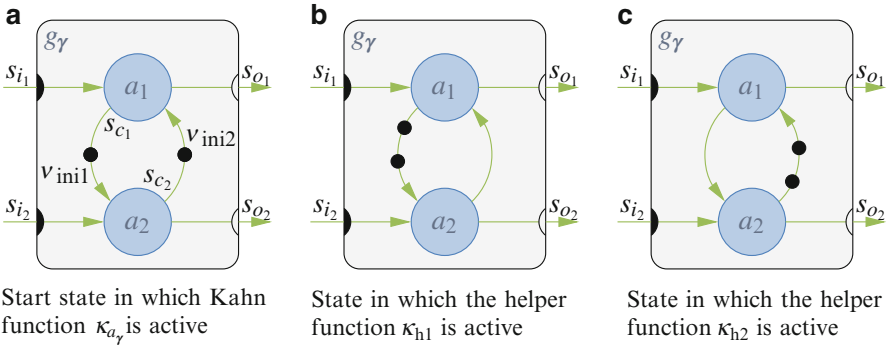


Fig. 3.6 Above (see (**a**)), the subgraph g_γ from Fig. 3.5 is depicted with the corresponding annotations to express it as a partial KPN. Moreover, in (**b**) and (**c**) the internal states encountered in the application of Kahn function κ_{a_γ} are illustrated

The behavior of a KPN is given by the *least fixed point* of an equation system that represents the connections of these actors to each other, e.g., as given in Equations (3.3) and (3.4) for the topology shown in Fig. 3.6a. The connection between actors a_1 and a_2 is provided by the signal s_{c_1} (see Equation (3.3)) produced by actor a_1 and the signal s_{c_2} (see Equation (3.4)) produced by actor a_2. The initial tokens v_{ini1} and v_{ini2} on the channels c_1 and c_2 connecting these two actors are modeled by prepending the corresponding initial token value in front of the corresponding signal, e.g., $\langle v_{\text{ini1}} \rangle^\frown s_{c_1}$ prepends the value v_{ini1} in front of the values carried by the signal s_{c_1}.

$$(s_{o_1}, s_{c_1}) = \kappa_{a_1}(s_{i_1}, \langle v_{\text{ini2}} \rangle^\frown s_{c_2}) \tag{3.3}$$

$$(s_{o_2}, s_{c_2}) = \kappa_{a_2}(s_{i_2}, \langle v_{\text{ini1}} \rangle^\frown s_{c_1}) \tag{3.4}$$

It turns out that such an equation system, e.g., Equations (3.2), (3.3), and (3.4), has a least fixed point for any input signal [30], and the function mapping the input signal to the corresponding fixed point solution of the equation system again represents a Kahn function. In that sense, the KPN model—in contrast to DDF and all MoCs of lower expressive power—is compositional. Another important characteristic emerges from this fact. The behavior of a *KPN model and all MoCs of lower expressive power*, be it a complete graph or a subgraph, is independent from the scheduling of actors, which is given by the operational implementation. Such data-flow models are called *sequence determinate* [30]. However, due to the non-compositionality of DDF, the resulting Kahn function is not generally representable via the operational semantics of DDF.

To exemplify, the Kahn function κ_{a_γ}—that is, the expression of the composite actor a_γ as a Kahn function—for the least fixed point of the Equations (3.2), (3.3), and (3.4) is given below. This function is defined via the main Kahn function κ_{a_γ} (given in Equation (3.5)) and the two helper functions κ_{h1} and

κ_{h2} (given in Equations (3.6) and (3.7)) that recursively call each other. The main Kahn function κ_{a_γ} is active in the state shown in Fig. 3.6a and can call either helper function κ_{h1} or κ_{h2} depending on whether a token is present at the head of either the input signal s_{i_1} or the input signal s_{i_2}, respectively. This will lead to the states depicted in Fig. 3.6b, c. From these states, via a call to the main Kahn function κ_{a_γ}, a transition back to the start state is performed if a token is present at the head of input signal s_{i_2} (for helper function κ_{h1}) or input signal s_{i_1} (for helper function κ_{h2}).

$$\kappa_{a_\gamma}(s_{i_1}, s_{i_2}) = \begin{cases} (\langle \mathbf{hd}(s_{i_1}) \rangle, \langle\rangle) \overrightarrow{\frown} \kappa_{\mathrm{h1}}(\mathbf{tl}(s_{i_1}), s_{i_2}) & \text{if } \#s_{i_1} \geq 1 \\ (\langle\rangle, \langle \mathbf{hd}(s_{i_2}) \rangle) \overrightarrow{\frown} \kappa_{\mathrm{h2}}(s_{i_1}, \mathbf{tl}(s_{i_2})) & \text{if } \#s_{i_2} \geq 1 \\ (\langle\rangle, \langle\rangle) & \text{otherwise} \end{cases} \tag{3.5}$$

$$\text{where } \kappa_{\mathrm{h1}}(s_{i_1}, s_{i_2}) = \begin{cases} (\langle\rangle, \langle \mathbf{hd}(s_{i_2}) \rangle) \overrightarrow{\frown} \kappa_{a_\gamma}(s_{i_1}, \mathbf{tl}(s_{i_2})) & \text{if } \#s_{i_2} \geq 1 \\ (\langle\rangle, \langle\rangle) & \text{otherwise} \end{cases} \tag{3.6}$$

$$\text{where } \kappa_{\mathrm{h2}}(s_{i_1}, s_{i_2}) = \begin{cases} (\langle \mathbf{hd}(s_{i_1}) \rangle, \langle\rangle) \overrightarrow{\frown} \kappa_{a_\gamma}(\mathbf{tl}(s_{i_1}), s_{i_2}) & \text{if } \#s_{i_1} \geq 1 \\ (\langle\rangle, \langle\rangle) & \text{otherwise} \end{cases} \tag{3.7}$$

Data-flow models with higher expressiveness are of the Non-Determinate Data Flow (NDF) MoC. This model as well as all previously discussed data-flow MoCs may be specified in the SystemC-based actor-oriented language SysteMoC that will be introduced in the next section.

3.3 Informal Introduction to SysteMoC

In this section, the SysteMoC modeling language [11, 12], a class library based on SystemC, will be presented. In SysteMoC parlance, data-flow graphs are called *network graphs*. As a running example, a network graph (see Fig. 3.7) of an application implementing Newton's iterative square root algorithm will be used throughout this section. Network graphs are very similar to DFGs as introduced in Definition 1, but are bipartite graphs consisting of channels $c \in C$ and actors $a \in A$. These vertices are connected via point-to-point connections between a channel and either an actor input or an output port. This acknowledges the fact that actors and channels must both be realized by some kind of resource and, hence, during DSE [25] a binding of vertices to resources of an architecture has to be explored. The *network graph* g_{sqr} implements Newton's iterative algorithm for calculating the square roots of an infinite input sequence generated by the `Src` actor a_1. Here, the square root values are computed by Newton's iterative algorithm realized via the `SqrLoop` actor a_2 performing error bound checking and the actors a_3 - a_4 performing an approximation step. After satisfying the error bound, the result is transported to the `Sink` actor a_5.

In the following, we will learn how to realize a network graph by instantiating actors and FIFO channels as well as connecting these FIFO channels with the ports

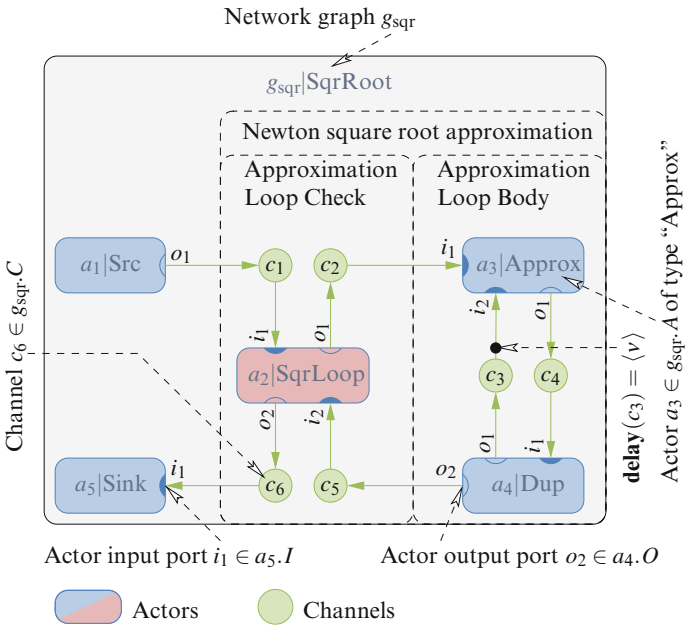


Fig. 3.7 The *network graph* g_{sqr} [10] displayed above implements Newton's iterative algorithm for calculating square roots. Actors are *bordered dark blue* and *shaded* according to their MoC (see Fig. 3.1 on page 62), while channels are depicted in *green*

of the actors. Next, in Sect. 3.3.2, we will study how to write the actor classes that are instantiated in the previous step. Finally, in Sect. 3.3.3, it will be shown how to specify the consumption and production rates exhibited by these actors via usage of so-called *actor Finite-State Machines (FSMs)*.

3.3.1 Specification of the Network Graph

SysteMoC is an open source C++ class library. Hence, all components mentioned in the previous section are represented by C++ classes. As an example, we will specify the `SqrRoot` C++ class that corresponds to the g_{sqr} DFG from Fig. 3.7 in Listing 1. All C++ classes representing DFGs must be derived from the `smoc_graph` base class provided by the SysteMoC library, e.g., as is shown in Line 1. All actors of a DFG must be instantiated in the constructor of the corresponding class, e.g., the lines colored dark blue as is shown in Lines 3 to 7 declaring the actor member variables that are instantiated in the constructor in Line 11. Subsequently, in the body of the constructor, e.g., Lines 12 to 20, which are colored green, the FIFO channels $c_1, c_2 \ldots c_6$ are connected to the input and output ports of these actors via the `connectNodePorts` method of SysteMoC.

In the simplest case (Lines 12 to 15), the `connectNodePorts` method takes two arguments: the output port o, from where the channel starts, and the input

Listing 1 Corresponding `SqrRoot` [10] class for the network graph g_{sqr}

```
1  class SqrRoot: public smoc_graph {
2  protected:
3    Src       a1;
4    SqrLoop   a2;
5    Approx    a3;
6    Dup       a4;
7    Sink      a5;
8  public:
9    SqrRoot(sc_module_name name)
10     : smoc_graph(name),
11       a1("a1"), a2("a2"), a3("a3"), a4("a4"), a5("a5") {
12     connectNodePorts(a1.o1, a2.i1); // c1
13     connectNodePorts(a2.o1, a3.i1); // c2
14     connectNodePorts(a2.o2, a5.i1); // c6
15     connectNodePorts(a4.o2, a2.i2); // c5
16     connectNodePorts(a3.o1, a4.i1,  // c4
17       smoc_fifo<double>(2));        // size(c4) = 2
18     connectNodePorts(a4.o1, a3.i2,  // c3
19       smoc_fifo<double>(3)          // size(c3) = 3
20         << 2.0);                    // delay(c3) = ⟨2.0⟩
21   }
22 };
```

port i, which is the destination of the channel. However, it is also possible to explicitly parameterize the created channel c with its channel capacity of **size**(c) tokens and a sequence of initial tokens **delay**(c). To exemplify, consider Lines 17 and 19, where a third parameter, the *channel initializer* `smoc_fifo<T>(n) « initial tokens...`, is given to the method `connectNodePorts`. If no channel initializer is given, then a FIFO channel with a channel capacity of one token and without any initial tokens is created between the output port o and the input port i. If a channel initializer is given, then it must be parameterized with the data type `T` carried by the channel and the channel capacity `n` in units of tokens. If initial tokens are required, these can be provided to the channel initializer via a sequence of "«"-operators each followed by an initial token, e.g., $\ll \nu_1 \ll \nu_2 \ll \ldots \nu_n$. Consider Line 17 as an example of how to specify a FIFO channel with a channel capacity of two tokens. In this case, the channel initializer is parameterized with the C++ `double` data type, denoting that the token values carried by the channel will be of the C++ `double` data type. An example for providing an initial token is given in Line 20, where the "«"-operator is used to provide the `double` value 2.0 as an initial token for the created channel between the ports $a_4.o_1$ and $a_3.i_2$.

3.3.2 Specification of Actors

Each actor in SysteMoC is represented by a C++ class, e.g., the class `SqrLoop` as defined in the following Listing 2 is representing the actor a_2 shown in Fig. 3.7,

that is derived from the `smoc_actor` base class (see Listing 2 Line 1) provided by the SysteMoC library. The input and output ports of an actor are specified by member variables of type `smoc_port_in` and `smoc_port_out` as exemplified in Listing 2 Lines 3 and 4, respectively. (Standard SystemC FIFO ports could not be used as the semantics of SysteMoC FIFOs extends standard FIFO semantics by the concept of a *random-access region* as shown in Fig. 3.9.) Furthermore, actors can have member variables, e.g., the variable `v` declared in Line 6. The functionality of an actor is represented by methods of the class, e.g., for the `SqrLoop` actor, the Lines 8 to 11 in Listing 2. Moreover, these methods are distinguished into *actions* (colored cyan), which can modify member variables, and *guards* (colored brown), which are declared as `const` methods and, hence, are not allowed to update the member variables. Later, in Sect. 3.3.3, it will be discussed how these action and guard methods are used as part of the actor FSM.

The `copyStore` method is responsible for forwarding the input token value on input port i_1 to the output port o_1 and storing the value into the member variable `v` for later replication on the output port o_1 by the method `copyInput`. This replication is required if the achieved accuracy by one square root iteration step by actor a_3 is below the bound determined by the guard `check`. On the other hand, if the approximation is within the error bound, then the action `copyApprox` is executed to forward this approximation to the output port o_2.

Note that actions themselves do not control token production or consumption, but only read or write values of input or output tokens via the syntax `i[n]` and `o[m]` that is used in Listing 2 and Fig. 3.9 to access the `n`th token relative to the read pointer, respectively, the `m`th token relative to the write pointer of the ring buffer (see Fig. 3.9) that realizes the channel that is connected to the corresponding port. The communication behavior, i.e., token production and consumption, is controlled solely by the actor FSM as will be detailed next.

Listing 2 SysteMoC implementation of the SqrLoop actor a_2

```
1  class SqrLoop: public smoc_actor {
2  public:
3    smoc_port_in<double>  i1, i2;
4    smoc_port_out<double> o1, o2;
5  private:
6    double v;
7
8    void copyStore()   {o1[0] = v = i1[0];}
9    void copyInput()   {o1[0] = v;        }
10   void copyApprox()  {o2[0] = i2[0];    }
11   bool check() const {return fabs(v-i2[0]*i2[0])<0.01;}
12
13   smoc_firing_state start, loop;
14 public:
15   SqrLoop(sc_module_name name);
16 };
```

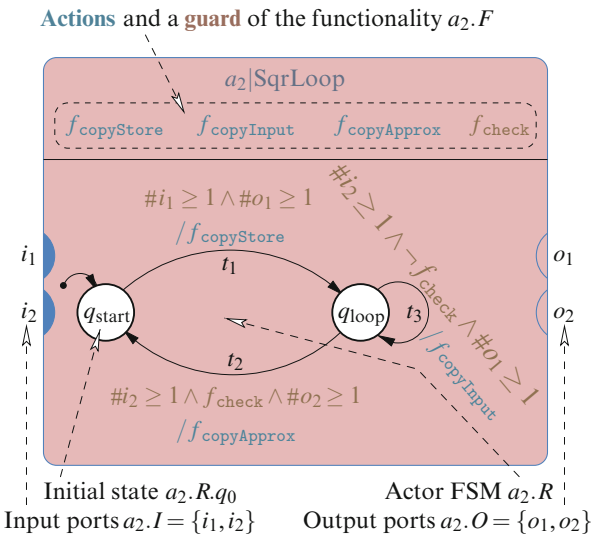


Fig. 3.8 The `SqrLoop` actor [10] from the network graph shown in Fig. 3.7 is composed of a set of *input ports* I and a set of *output ports* O, its *functionality* F, and its *FSM* R. The functionality can be further subdivided into actions (*colored cyan*) and guards (*colored brown*) as used by the FSM

3.3.3 Specification of the Communication Behavior

The *communication behavior* of an actor is separated strictly from the functional behavior of the actor on purpose. The communication behavior describes how and under what condition tokens are consumed and produced by an actor. In SysteMoC, the abstraction is done by representation of the communication behavior of an actor by its actor FSM, e.g., as seen in Fig. 3.8. More formally, a SysteMoC actor can be defined as follows:

Definition 2 (Actor [11]). An actor is a tuple $a = (I, O, F, R)$ containing a set of *actor input ports* I and *actor output ports* O, the *actor functionality* $F = F_{action} \cup F_{guard}$ partitioned into a set of actions and a set of guards, as well as the *actor FSM* R that is determining the *communication behavior* of the actor.

To exemplify, the methods `copyStore`, `copyInput`, and `copyApprox` (see Lines 8 to 10 in Listing 2) correspond to their respective actions, i.e., $f_{\text{copyStore}}$, $f_{\text{copyInput}}$, $f_{\text{copyApprox}} \in F_{action}$. Finally, the guard $f_{\text{check}} \in F_{guard}$ is represented by the `const` method `check` declared in Line 11.

For example, in state q_{start}, transition t_1 may be taken if there exists at least one token on input port i_1 ($\#i_1 \geq 1$) and at least one free place on the FIFO connected to output port o_1 ($\#o_1 \geq 1$). Once this condition is fulfilled, t_1 is taken and the action $f_{\text{copyStore}}$ executed. The transition or *actor firing* ends with the consumption of one token from i_1 and production of one token at the output port o_1 with the value computed according to the function $f_{\text{copyStore}}$ shown in Listing 2. The actor FSM

Listing 3 SysteMoC implementation of the SqrLoop actor a_2

```
1  class SqrLoop: public smoc_actor {
2    ...
3    smoc_firing_state start, loop;
4  public:
5    SqrLoop(sc_module_name name)
6      : smoc_actor(name, start),
7        i1("i1"), i2("i2"), o1("o1"), o2("o2"),
8        start("start"), loop("loop")
9    {
10     start =
11         i1(1)                                    >>
12         o1(1)                                    >>
13         SMOC_CALL(SqrLoop::copyStore)            >> loop
14       ;
15     loop  =
16         (i2(1) &&  SMOC_GUARD(SqrLoop::check)) >>
17         o2(1)                                    >>
18         SMOC_CALL(SqrLoop::copyApprox)           >> start
19       |
20         (i2(1) && !SMOC_GUARD(SqrLoop::check)) >>
21         o1(1)                                    >>
22         SMOC_CALL(SqrLoop::copyInput)            >> loop
23       ;
24   }
25 };
```

of the `SqrLoop` actor as depicted in Fig. 3.8 is constructed by the code shown in Listing 3 Lines 10 to 23.

The states Q of the actor FSM themselves are represented by member variables of type `smoc_firing_state`, e.g., the state `start` and `loop` in Line 3, that is $Q = \{q_{\text{start}}, q_{\text{loop}}\}$. The initial state of the actor FSM is determined by providing the base class `smoc_actor` with the corresponding state. For the `SqrLoop` actor, the initial state q_0 is set in Line 6 to q_{start} (`start`).

The syntax for naming SystemC and also SysteMoC entities is demonstrated in Lines 7 and 8, where the actor input and output ports as well as the states of the actor FSM are named, respectively. However, in the interest of conciseness, it will be assumed that all SystemC/SysteMoC entities are named like their declarations in the source code shown in the following examples, but this naming will not be shown explicitly.

The transition t_1 from q_{start} to q_{loop} is given in Lines 11 to 13. Here, guards are again colored brown while actions are colored cyan. In detail, we use the syntax `i(n)` and `o(m)` to express the condition that at least n tokens, respectively, at least m free places must be present on the channel connected to the input port i and the channel connected to the output port o. Methods of the class used as actions or guard functions must be marked via usage of the `SMOC_CALL` or

SMOC_GUARD macros, respectively. Transitions t_2 and t_3 are defined accordingly in Lines 16 to 18 and Lines 20 to 22. As can be seen, SysteMoC—in contrast to KPN [24] and FunState (The model was initially published [42] as State Machine Controlled Flow Diagrams (SCF), but in later literature it is referred to as FunState, which is a short for Functions Driven by State Machines.) [41]—distinguishes its functions further into actions $f_{action} \in F_{action}$ and guards $f_{guard} \in F_{guard}$. To exemplify, the SqrLoop actor depicted in Fig. 3.8 is considered. This actor has three actions $F_{action} = \{ f_{\text{copyStore}}, f_{\text{copyInput}}, f_{\text{copyApprox}} \}$ and one guard function $F_{guard} = \{ f_{\text{check}} \}$. Naturally, the guard f_{check} is only used in a transition guard of the FSM R while the actions of the FSM are drawn from the set of actions F_{action}.

In SysteMoC, each FSM state is defined explicitly by an assignment of a list of outgoing transitions. To exemplify, the state q_{start} is defined in Listing 3 Line 10 by assigning transition t_1 (Lines 11 to 13) to it. If a state has multiple outgoing transitions, e.g., state q_{loop} with the transitions t_2 (Lines 16 to 18) and t_3 (Lines 20 to 22), then the outgoing transitions are joined via the "|"-operator (Line 19) and assigned to the state variable (Line 15).

3.4 Semantics and Execution Behavior of SysteMoC

More formally, an actor FSM resembles the FSM definition from FunState [41], but with slightly different definitions for actions and guards and defined as follows:

Definition 3 (Actor FSM [11]). The FSM R of an actor a is a tuple (Q, q_0, T) containing a finite set of *states* Q, an *initial state* $q_0 \in Q$, and a finite set of *transitions* T. A *transition* $t \in T$ itself is a tuple $(q_{\text{src}}, k, f_{action}, q_{\text{dst}})$ containing the source state $q_{\text{src}} \in Q$, from where the transition is originating, and the destination state $q_{\text{dst}} \in Q$, which will become the next current state after the execution of the transition starting from the current state q_{src}. Furthermore, if the transition t is taken, then an action f_{action} from the set of functions of the *actor functionality* $a.F_{action}$ will be executed. (We use the "."-operator, e.g., $a.F_{action}$, for member access of tuples whose members have been explicitly named in their definition, e.g., member F_{action} of the actor a from Definition 2.) Finally, the execution of the transition t itself is guarded by the *guard* k.

A transition t will be called an *outgoing transition* of a state q if and only if the state q is the source state $t.q_{\text{src}}$ of the transition. Correspondingly, a transition will be called an *incoming transition* of a state q if and only if the state q is the destination state $t.q_{\text{dst}}$ of the transition. Furthermore, a transition is *enabled* if and only if its guard k evaluates to **t** and it is an outgoing transition of the current state of the actor.

An actor is enabled if and only if it has at least one enabled transition. The firing of an actor corresponds to a non-deterministic selection and execution of one transition out of the set of enabled transitions of the actor. In general, if multiple

actors have enabled transitions, then the transition, and hence its corresponding actor, is chosen non-deterministically out of the set of all enabled transitions in all actors. In summary, the execution of a SysteMoC model can be divided into three phases:

- Determine the set of enabled transitions by checking each transition in each actor of the model. If this set is empty, then the simulation of the model will terminate.
- Select a transition t from the set of enabled transitions, and fire the corresponding actor by executing the selected transition t, thus computing the associated action f_{action}.
- Subsequently, consume and produce tokens as encoded in the selected transition t. This might enable new transitions. Go back to the first step.

In contrast to FunState [41], a guard k is more structured. Moreover, it is partitioned into the following three components:

- The *input guard* which encodes a conjunction of input predicates on the number of available tokens on the input ports, e.g., $\#i_1 \geq 1$ denotes an input predicate that tests if at least one token is available at the actor input port i_1. Hence, consumption rates can be associated with each transition by the **cons** : $(T \times I) \rightarrow \mathbb{N}_0$ function that determines for each transition and input port/channel combination the number of tokens that have to be at least present to enable the transition.
- The *output guard* which encodes a conjunction of output predicates on the number of free places on the output ports, e.g., $\#o_1 \geq 1$ denotes an output predicate that tests if at least one free place is available at the actor output port o_1. Thus, production rates can be associated with each transition by the **prod** : $(T \times O) \rightarrow \mathbb{N}_0$ function that specifies for each transition and output port/channel combination the number of free places that must be at least available to enable the transition.
- The *functionality guard* which encodes a logical composition of guard functions of the actor, e.g., $\neg f_{\texttt{check}}$ annotated to the transition t_3 of the actor FSM of the actor a_2 in Fig. 3.8. Hence, the *functionality guard* depends on the *functionality state* and the token values on the input ports.

Here, the notion of a *functionality state* $q_{\text{func}} \in Q_{\text{func}}$ of an actor is used. The functionality state is an abstract representation of the C++ variables in the real implementation of a SysteMoC actor. This functionality state is required for notational completeness, that is for the formal definitions of action and guard functions, i.e., $f_{action} : Q_{\text{func}} \times S^{|I|} \rightarrow Q_{\text{func}} \times S^{|O|}$ and $f_{guard} : Q_{\text{func}} \times S^{|I|} \rightarrow \{\mathbf{t}, \mathbf{f}\}$. (The usual notation $|X|$ is used to denote the cardinality of a set X.) Both types of functions depend on the functionality state of their actor. Furthermore, an action may update this state, while a guard function may not. Hence, all SysteMoC

actor firings are sequentialized over this functionality state. Therefore, multiple actor firings of the same actor cannot be executed in parallel. That is, in data-flow parlance, all SysteMoC actors have a virtual self-loop with one initial token that corresponds to $q_{\text{func}} \in Q_{\text{func}}$. However, as the functionality state is not used for any optimization and analysis algorithms presented in this chapter, it has been excluded from Definition 2.

Note that the consumption and production rate functions also specify the number of tokens that are consumed/produced on the input/output ports if a transition t is taken. That is, $\forall i \in I : \mathbf{cons}(i, t.f_{action}) = \mathbf{cons}(t, i) \wedge \forall f_{guard}$ contained in $t.k : \mathbf{cons}(i, f_{guard}) \leq \mathbf{cons}(t, i)$ and $\forall o \in O : \mathbf{prod}(t.f_{action}, o) = \mathbf{prod}(t, o)$. Hence, for a SysteMoC model to be *well formed*, the number of tokens accessed on the different input and output ports by an action $t.f_{action}$ associated with the transition t as well as the guard functions f_{guard} used in the transition guard $t.k$ has to conform to the consumption and production rates of the transition.

We enforce this requirement by checking that access to tokens by actions and guards via the ports, e.g., the syntax `i[n]` and `o[m]` that is used in Fig. 3.9, Listing 2 to access the `n`th token relative to the read pointer, respectively, the `m`th token relative to the write pointer, does not access tokens outside the so-called *random-access region* that is controlled by the actor FSM as will be explained in the following. Hence, as only tokens inside the random-access region can be modified by the actions or read by the guards, the actor FSM fully controls the communication behavior of the actor.

To exemplify, we consider a simple example with just one source and one sink actor connected by a single channel as depicted in Fig. 3.9. Here, the random-access regions are marked by bold brown-bordered boxes.

Both actors a_1 and a_2 are in the process of executing their respective actions f_{sink} and f_{src}. Thus, the guards $\#i_1 \geq 3$ and $\#o_1 \geq 2$ are satisfied, but the three tokens and two free places are not yet consumed and produced, respectively, but only reserved for consumption and production when execution of the actions finishes. Then, the read and write pointers of the FIFO channels will also be advanced by the actor FSMs by three, respective, two, tokens. Before the execution of the actions is started, the random-access regions are sized according to the guards of the transitions that are currently taken. During the execution of the actions, random access and update of all data values of the tokens contained in the random-access regions—but not outside these regions—is allowed by the executing actions.

Finally, for a SysteMoC model to be well formed, no sharing of state between actors via global variables is allowed. This requirement cannot be enforced by the SysteMoC library itself and is also usually not a problem in the code of the actor itself, but in library code that is used by the actor. A discussion of modeling library dependencies via *library tasks* and the problem of persistent states inside these libraries is tackled by the Common Intermediate Code (CIC) model that is introduced in ▸ Chap. 29, "HOPES: Programming Platform Approach for Embedded Systems Design".

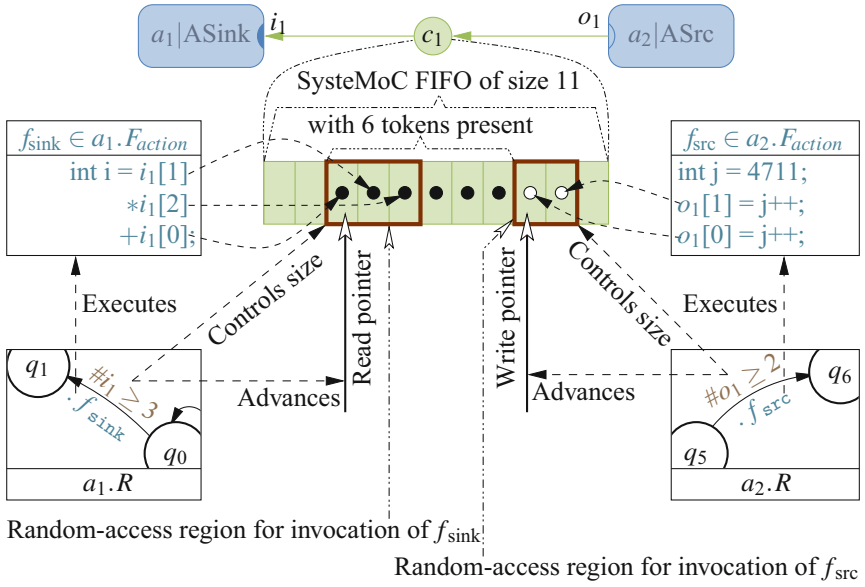


Fig. 3.9 Depicted above is a simple source-sink example that is used to explain the semantics of a SysteMoC FIFO associated with the channel c_1. The buffer memory of the channel c_1 is organized as a *ring buffer* with its associated *read and write pointers*. The buffer memory has a capacity of 11 tokens (depicted as *light green cells* inside the *green-bordered box* representing the buffer memory) and is already filled with 6 tokens (the 6 *solid black dots*). From the remaining five free places, two places have been reserved (the two *black-bordered* but *white-filled dots*) by the actor FSM $a_2.R$ for the execution of the action f_{src}. These reserved places correspond to tokens that have not yet been associated with data values v. It is the responsibility of the action to compute the missing data values. Finally, for each function invocation and for each channel accessed by the invoked function, a random-access region is defined for accessing the channel. Hence, two random-access regions (depicted as *bold brown-bordered boxes*) are shown in the above figure

3.5 Analysis of SysteMoC Models

SysteMoC permits modeling of non-determinate data-flow graphs. However, when using SysteMoC models as an input to Hardware/Software Codesign (HSCD) flows, it is advantageous for reason of analyzability to identify parts of the model, which belong to restricted MoCs. Thanks to its formal representation of the firing behavior of an actor by an actor FSM, MoC identification can be performed within SysteMoC for its actors. In the following, we present the representation as well as identification of SDF and CSDF actors. The ability of classification [12,45] serves as a distinction between SysteMoC and Ptolemy [8,23,35] as well as ForSyDe, which is introduced in ▸ Chap. 4, "ForSyDe: System Design Using a Functional Language and Models of Computation", and similar approaches that use distinct modeling libraries for each different MoC. For example, in Ptolemy each actor is associated with a director which explicitly specifies the MoC under which the actor is executed. This is

important as the data-flow model of SysteMoC is chosen in order to provide greatest expressiveness. Hence, the analyzability of a general SysteMoC specification (e.g., with respect to deadlock freedom or the ability to be executed in bounded memory) is limited. Nonetheless, only very simple actors will be classified as belonging to the static data-flow domain. Hence, a more detailed discussion as well as identification of BDF and DDF actors can be found in [44]. Scheduling algorithms that take advantage of this information are provided in [5] for the BDF MoC. Optimizations for more expressive MoCs is an ongoing topic of current research.

3.5.1 Representing SDF and CSDF Actors in SysteMoC

As SysteMoC permits modeling of quite general types of data-flow graphs, it is also possible to represent any SDF and CSDF actor in SysteMoC. Given a CSDF graph as defined in Definition 1 and using the ancillary definitions of the **cons** and **prod** functions as described in Sect. 3.2.2.3, then for a given CSDF actor a, a functionally equivalent SysteMoC actor a' can be constructed as follows: First, the input ports I and output ports O of the SysteMoC actor a' correspond to the incoming and outgoing channels of the CSDF actor a, respectively. Assuming that this actor has τ CSDF phases, then the corresponding actor FSM R is built by generating a state for each phase of the CSDF actor, i.e., $Q = \{q_0, q_1, \ldots q_{\tau-1}\}$, and marking the state q_0 as the initial state of the FSM. Next, these states are connected such that the resulting transitions form a single cycle, i.e., $T = \{(q_0, k_0, f_{\texttt{phase}_0}, q_1), (q_1, k_1, f_{\texttt{phase}_1}, q_2), \ldots (q_{\tau-1}, k_{\tau-1}, f_{\texttt{phase}_{\tau-1}}, q_0)\}$. As an example, the resulting FSM R of the CSDF actor a_3 is shown in Fig. 3.10b. Here, the actions $f_{\texttt{phase}_0}, f_{\texttt{phase}_1}, \ldots f_{\texttt{phase}_{\tau-1}}$ are associated with the firing of the CSDF actor in the corresponding phase. The token consumption and production rates encoded by the guards $k_0, k_1, \ldots k_{\tau-1}$ and given by the functions **cons** : $(T \times I) \rightarrow \mathbb{N}_0$ and **prod** : $(T \times O) \rightarrow \mathbb{N}_0$ of the SysteMoC model can be directly derived from the functions **cons** : $C \rightarrow \mathbb{N}_0^\tau$ and **prod** : $C \rightarrow \mathbb{N}_0^\tau$ of the CSDF graph. Remember that we use the notion of ports interchangeably with the channels connected to these ports. Hence, we can define the **cons** and **prod** functions of the SysteMoC model as follows:

$$\textbf{prod}(t_l, o) = n_l \quad \text{where } (n_0, n_1, \ldots n_{\tau-1}) = \textbf{prod}(o) \quad \forall l \in \{0, 1, \ldots \tau - 1\}, o \in O$$

$$\textbf{cons}(t_l, i) = m_l \quad \text{where } (m_0, m_1, \ldots m_{\tau-1}) = \textbf{cons}(i) \quad \forall l \in \{0, 1, \ldots \tau - 1\}, i \in I$$

To exemplify, for the CSDF actor a_3 from Fig. 3.10c, the resulting normalized actor FSM is shown in Fig. 3.10b. Corresponding to the first phase, transition t_0 consumes one token from i_1 and produces seven tokens on o_1. The second phase embodied by transition t_1 consumes two tokens from i_1 and produces one token on o_1. The remaining phases are derived accordingly.

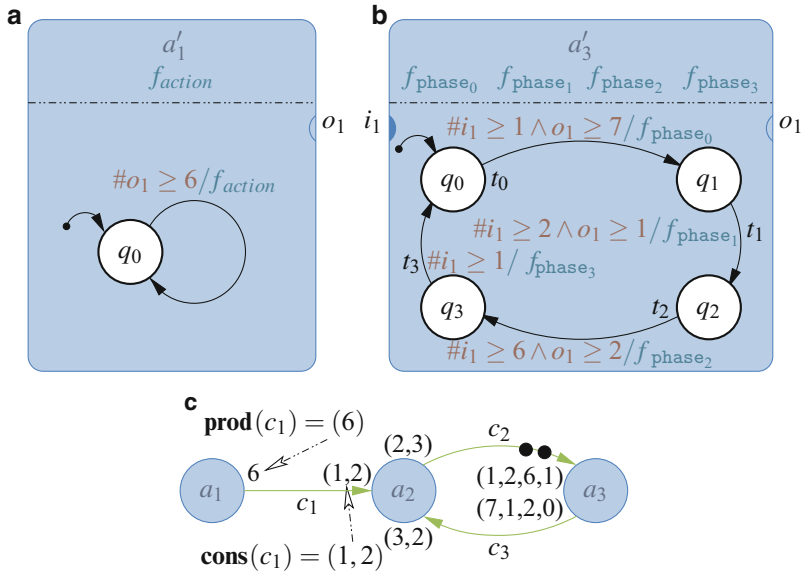


Fig. 3.10 Example translations of an SDF and a CSDF actor into SysteMoC. (**a**) SysteMoC realization of the SDF actor a_1 from the DFG below. (**b**) SysteMoC realization of the CSDF actor a_3 from the DFG below. (**c**) Example of a CSDF graph also containing an SDF actor

3.5.2 SDF and CSDF Semantics Identification for SysteMoC Actors

In order to be able to apply MoC-specific analysis methods such as deadlock analysis or generation of static schedules, it is important to recognize individual actors or subgraphs thereof belonging to well-known data-flow models of computation such as HSDF, SDF, and CSDF. While it is possible to transform any static data-flow actor into a SysteMoC actor as described above, it is much more difficult to check if a given SysteMoC actor shows SDF or CSDF semantics. It will be shown that this can be accomplished by inspection and analysis of the *actor FSM* of a SysteMoC actor.

The most basic representation of static actors encoded as actor FSMs can be seen in Fig. 3.10. To exemplify, the SDF actor depicted in Fig. 3.10a contains only one transition, which produces six tokens onto the actor output port o_1. Clearly, the actor exhibits a static communication behavior corresponding to the SDF MoC. However, for the classification algorithm to be able to ignore the guard functions used by an actor FSM, certain assumptions have to be made. It will be assumed that given sufficient tokens and free places on the actor input and output ports, at least one of the guards of the outgoing transitions will evaluate to true, and, hence, there exists at least one enabled outgoing transition. This is required in order to be able to activate the equivalent SDF or CSDF actor an infinite number of times. It will also be assumed that an actor will consume or produce tokens every now

and then. These assumptions are not erroneous as, otherwise, there exists an infinite loop in the execution of the actor where a cycle of the FSM is traversed and each transition in this cycle neither consumes nor produces any tokens. This can clearly be identified as a bug in the implementation of the actor, similar to an action of a transition that never terminates. For the exact mathematical definition of the above-given requirements, see [45].

The idea of the classification algorithm is to check if the communication behavior of a given actor can be reduced to a basic representation, which can be easily classified into the SDF or CSDF MoC. Note that the basic representations for both SDF and CSDF models are FSMs with a single cycle, e.g., as depicted for the CSDF actor in Fig. 3.10b.

Yet, not all *actor FSMs* satisfy this condition (see Fig. 3.11). However, some of them, e.g., Fig. 3.11b, c, still show the communication behavior of a static actor. It can be distinguished if the analysis of the *actor functionality state* (see Definition 2) is required to decide whether an actor is a static actor, e.g., Fig. 3.11c, or not, e.g., Fig. 3.11b. In the following, the actor functionality state will not be considered. Therefore, the presented classification algorithm will fail to classify Fig. 3.11c as a static actor, but will achieve the classification as static for the actor shown in Fig. 3.11b. In that sense, the algorithm only provides a *sufficient* criterion for the problem of static actor detection. A *sufficient* and *necessary* criterion cannot be given as the problem is undecidable in the general case.

The algorithm starts by deriving a set of *classification candidates* solely on the basis of the specified actor FSM. Each candidate is later checked for consistency with the entire FSM state space via Algorithm 1 that will be explained in detail later in this section. If one of the *classification candidates* is accepted by Algorithm 1, then the actor is recognized as a CSDF actor where the CSDF phases are given by the accepted *classification candidate*.

Definition 4 (Classification Candidate [45]). A possible CSDF behavior of a given actor is captured by a *classification candidate* $\boldsymbol{\mu} = \langle \mu_0, \mu_1, \ldots \mu_{\tau-1} \rangle$ where each $\mu = (\text{cons}, \text{prod})$ represents a phase of the CSDF behavior and τ is the number of phases.

To exemplify, Fig. 3.10b is considered. In this case, four phases are present, i.e., $\tau = 4$, where the first phase (μ_0) consumes one token from i_1 and produces seven tokens on o_1, the second phase (μ_1) consumes two tokens from i_1 and produces one token on o_1, the third phase (μ_2) consumes six tokens and produces two, and the final phase (μ_3) only consumes one token.

It can be observed that all paths starting from the initial *actor FSM* state q_0 must comply with the *classification candidate*. Furthermore, as CSDF exhibits cyclic behavior, the paths must also contain a cycle. The classification algorithm will search for such a path $p = \langle t_1, t_2, \ldots, t_n \rangle$ of transitions t_i in the *actor FSM* starting from the initial state q_0. The path can be decomposed into an acyclic prefix path p_a and a cyclic path p_c such that $p = p_a \widehat{\ } p_c$, i.e., $p_a = \langle t_0, t_1, \ldots t_{l-1} \rangle$ being the

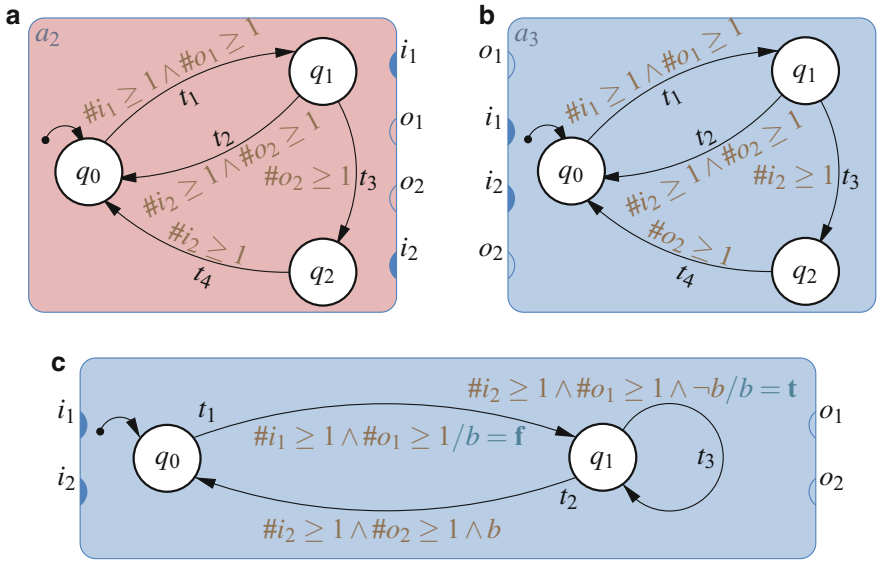


Fig. 3.11 Various actor FSMs that do not match the basic representation of static actors as exemplified in Fig. 3.10. (**a**) Visualized above is the actor a_2 with an *actor FSM* that cannot be converted to a basic representation of static actors. (**b**) Shown is the actor a_3 containing an *actor FSM* that can be converted to basic CSDF representation. (**c**) Shown is an actor that seems to belong to the *dynamic domain*, but exhibits CSDF behavior due to the manipulation of the Boolean variable b. Hence, leading to a cyclic execution of the transitions t_1, t_3, and t_2 of its *actor FSM*

prefix and $p_c = \langle t_l, t_{l+1}, \ldots t_{n-1} \rangle$ being the cyclic part, that is $t_l.q_{\text{src}} = t_{n-1}.q_{\text{dst}}$. After such a path p has been found, a set of *classification candidates* can be derived from the set of all nonempty prefixes $\{p' \sqsubseteq p \mid \#p' \in \{1, 2, \ldots n\}\}$ of the path p. (Here, the "$\sqsubseteq$"-operator denotes that a sequence is a prefix of some other sequence, e.g., $\langle 3, 1 \rangle \sqsubseteq \langle 3, 1 \rangle \sqsubseteq \langle 3, 1, 2 \rangle \sqsubseteq \langle 3, 1, 2, \ldots \rangle$, but $\langle 3, 1 \rangle \not\sqsubseteq \langle 4, 1, 2 \rangle$ as well as $\langle 3, 1 \rangle \not\sqsubseteq \langle 3 \rangle$.) A *classification candidate* $\boldsymbol{\mu} = \langle \mu_0, \mu_1, \ldots \mu_{\tau-1} \rangle$ is derived from a nonempty prefix p' by (1) unifying adjacent transitions t_j, t_{j+1} according to the below given *transition contraction condition* until no more contractions can be performed and (2) deriving the CSDF phases μ_j of the *classification candidate* from the transitions t_j of the contracted path computed in step (1).

Definition 5 (Transition Contraction Condition [12]). Two transitions t_j and t_{j+1} of a prefix path p' can be contracted if t_j only consumes tokens, i.e., $\mathbf{prod}(t_j, O) = \mathbf{0}$, or t_{j+1} only produces tokens, i.e., $\mathbf{cons}(t_{j+1}, I) = \mathbf{0}$. (For notational brevity, the construct $\mathbf{cons}(t, I) = (n_{i_1}, n_{i_2} \ldots, n_{i_{|I|}}) = \boldsymbol{n}_{\text{cons}}$ is used to denote the vector $\boldsymbol{n}_{\text{cons}}$ of numbers of tokens consumed by taking the transition. The equivalent notation $\mathbf{prod}(t, O)$ is also used similarly for produced tokens.) The resulting transition t' has the combined consumption and production rates given as $\mathbf{cons}(t', I) = \mathbf{cons}(t_j, I) + \mathbf{cons}(t_{j+1}, I)$ and $\mathbf{prod}(t', O) = \mathbf{prod}(t_j, O) + \mathbf{prod}(t_{j+1}, O)$.

Algorithm 1 Validation of a *classification candidate* $\boldsymbol{\mu}$ for the *actor FSM* R

function VALIDATECLASSCAND($\boldsymbol{\mu}$, R)
  $\tau = \#\boldsymbol{\mu}$ ▷ Number of CSDF phases τ
  $\langle \mu_0, \mu_1, \ldots \mu_{\tau-1} \rangle = \boldsymbol{\mu}$ ▷ CSDF phases $\mu_0, \mu_1, \ldots \mu_{\tau-1}$
  $\omega_0 = (0, \mu_0.\text{cons}, \mu_0.\text{prod})$ ▷ Initial annotation tuple
  `queue` $\leftarrow \langle\rangle$ ▷ Set up the empty `queue` for breadth-first search
  `ann` $\leftarrow \emptyset$ ▷ Set up the empty set of annotations
  `queue` $\leftarrow$ `queue`$^\frown(R.q_0)$ ▷ Enqueue $R.q_0$
  `ann` $\leftarrow$ `ann`$\cup\{ (R.q_0, \omega_0) \}$ ▷ Annotate the initial tuple
  while #`queue` > 0 **do** ▷ While the `queue` is not empty
    $q_{\text{src}} =$ **hd**(`queue`) ▷ Get head of `queue`
    $\omega_{\text{src}} =$ `ann`(q_{src}) ▷ Get annotation tuple for state q_{src}
    `queue` $\leftarrow$ **tl**(`queue`) ▷ Dequeue head from `queue`
    for all $t \in R.T$ where $t.q_{\text{src}} = q_{\text{src}}$ **do**
      if $\omega_{\text{src}}.\text{cons} \not\geq$ **cons**$(t, I) \vee \omega_{\text{src}}.\text{prod} \not\geq$ **prod**(t, O) **then**
        return f ▷ Reject $\boldsymbol{\mu}$ due to failed transition criterion I
      end if
      if $\omega_{\text{src}}.\text{cons} \neq$ **cons**$(t, I) \wedge$ **prod**$(t, O) \neq$ **0 then**
        return f ▷ Reject $\boldsymbol{\mu}$ due to failed transition criterion II
      end if
      $\omega_{\text{dst}} \leftarrow$ derive from ω_{src} and t as given by Equation (3.8)
      if $\exists \omega'_{\text{dst}} : (t.q_{\text{dst}}, \omega'_{\text{dst}}) \in$ `ann` **then** ▷ Annotated tuple present?
        if $\omega'_{\text{dst}} \neq \omega_{\text{dst}}$ **then** ▷ Check annotated tuple for consistency
          return f ▷ Reject classification candidate $\boldsymbol{\mu}$
        end if
      else ▷ No tuple annotate to state $t.q_{\text{dst}}$
        `ann` $\leftarrow$ `ann`$\cup\{ (t.q_{\text{dst}}, \omega_{\text{dst}}) \}$ ▷ Annotate tuple ω_{dst}
        `queue` $\leftarrow$ `queue`$^\frown(t.q_{\text{dst}})$ ▷ Enqueue $t.q_{\text{dst}}$
      end if
    end for
  end while
  return t ▷ Accept classification candidate $\boldsymbol{\mu}$
end function

For clarification, Fig. 3.11a, b are considered and it is assumed that the path $p = \langle t_1, t_3, t_4 \rangle$ has been found. Hence, the set of all nonempty prefixes is $\{ \langle t_1 \rangle, \langle t_1, t_3 \rangle, \langle t_1, t_3, t_4 \rangle \}$. In both depicted FSMs, the transition t_2 consumes and produces exactly as many tokens as the transition sequence $\langle t_3, t_4 \rangle$. However, the transition sequence $\langle t_3, t_4 \rangle$ in Fig. 3.11b can be contracted as t_3 only consumes tokens while transition sequence $\langle t_3, t_4 \rangle$ in Fig. 3.11a cannot be contracted. Hence, for Fig. 3.11b, the *classification candidate* $\boldsymbol{\mu} = \langle \mu_0, \mu_1 \rangle$ derived from the prefix $p' = \langle t_1, t_3, t_4 \rangle$ is as follows:

$$\begin{aligned}
\mu_0.\text{cons} &= \textbf{cons}(t_1, I) &&= (1, 0) \\
\mu_0.\text{prod} &= \textbf{prod}(t_1, O) &&= (1, 0) \\
\mu_1.\text{cons} &= \textbf{cons}(t_3, I) + \textbf{cons}(t_4, I) &&= (0, 1) \\
\mu_1.\text{prod} &= \textbf{prod}(t_3, O) + \textbf{prod}(t_4, O) &&= (0, 1)
\end{aligned}$$

On the other hand, for Fig. 3.11a, the *classification candidate* $\boldsymbol{\mu} = \langle \mu_0, \mu_1, \mu_2 \rangle$ derived from the prefix $p' = \langle t_1, t_3, t_4 \rangle$ does not exhibit any contractions and is depicted below as:

$$\begin{aligned} \mu_0.\text{cons} &= (1,0)\ \mu_0.\text{prod} = (1,0) \\ \mu_1.\text{cons} &= (0,0)\ \mu_1.\text{prod} = (0,1) \\ \mu_2.\text{cons} &= (0,1)\ \mu_2.\text{prod} = (0,0) \end{aligned}$$

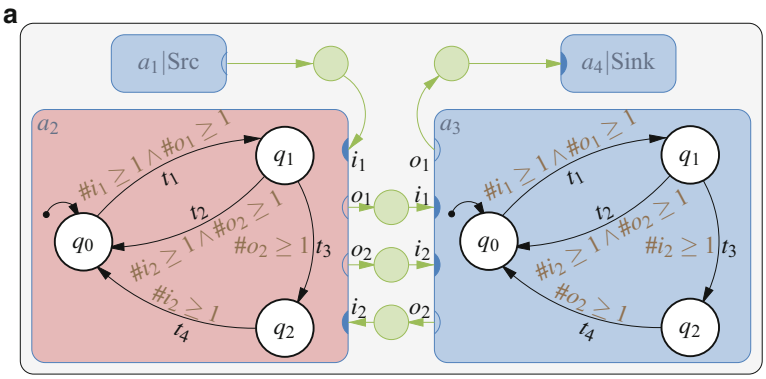
The underlying reasons for the *transition contraction condition* from Definition 5 are discussed in the following. To illustrate, the data-flow graph depicted in Fig. 3.12a which uses two actors a_2 and a_3 containing the FSMs from Fig. 3.11a, b is considered. Obviously, there exist dependencies between the transitions in a legal transition trace of the actors a_2 and a_3 as shown in Fig. 3.12b.

For example, it can be seen in Fig. 3.12c that if the transition sequence $\langle t_3, t_4 \rangle$ of actor a_3 is contracted into a single transition t_c, then the resulting dependencies of t_c are exactly the same as for transition t_2. This is the reason why the FSM from Fig. 3.11b can be classified into a CSDF actor. Furthermore, the contraction is a valid transformation as it does not change the set of possible transition sequences of the whole data-flow graph, apart from the substitution of the transition sequence $\langle t_3, t_4 \rangle$ by the transition t_c. This can be seen by comparing the possible transition sequences depicted in Fig. 3.12b, c. Compacting the transition sequence $\langle t_3, t_4 \rangle$ of actor a_3 generates the transition t_c, which is inducing the data dependency that the token produced on port o_2 by the transition t_c depends on the token consumed on port i_2 by the transition t_c. However, the previous transition sequence $\langle t_3, t_4 \rangle$ also induces this data dependency as t_4 can only be taken after t_3.

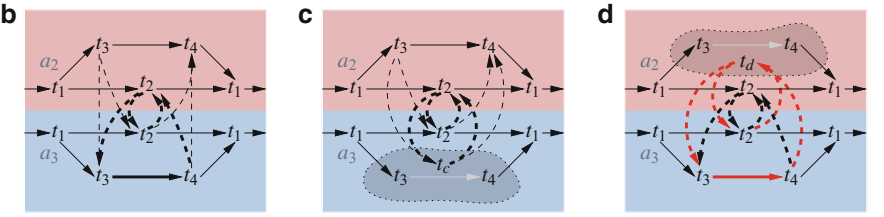
In contrast to this, the contraction of the transition sequence $\langle t_3, t_4 \rangle$ of actor a_2 into a transition t_d does introduce a new erroneous data dependency from the consumption of a token on i_2 to the production of a token on o_2. The data dependency is erroneous as the original transition sequence $\langle t_3, t_4 \rangle$ has no such dependency as it first produces the token on o_2 before trying to consume a token on i_2. Indeed, an erroneous contraction might introduce a deadlock into the system as can be seen in Fig. 3.12d where the transition t_d is part of two dependency cycles $a_2.t_d \rightarrow a_3.t_2 \rightarrow a_2.t_d$ and $a_2.t_d \rightarrow a_3.t_3 \rightarrow a_3.t_4 \rightarrow a_2.t_d$ which are not present in the original dependency structure depicted in Fig. 3.12b.

After the set of *classification candidates* has been derived from the actor FSM, each candidate is checked for validity by Algorithm 1. The checks are performed sequentially starting from the *classification candidate* derived from the shortest prefix p' to the candidate derived from the full path p. If a candidate is accepted by Algorithm 1, then the actor is a CSDF actor with phases as given by the accepted *classification candidate* $\boldsymbol{\mu}$.

The main idea of the validation algorithm is to check a *classification candidate* against all possible transition sequences reachable from the initial state of the actor FSM. However, due to the existence of contracted transitions in the *classification candidate* as well as in the actor FSM, a simple matching of a phase μ_j to



Shown is a DFG containing the actors a_2 and a_3 from Figure 3.11. Due to mutual dependencies between the actor firings of the actors a_2 and a_3, actor a_2 will never execute transition t_2. On the other hand, actor a_3 is free to choose either the transition sequence $\langle t_3, t_4 \rangle$ or the transition t_2 in its execution trace.



Shown are the dependencies in the transition trace of actor a_2 and a_3. Two dependency cycles are already present (marked bold black) that cause the inability to fire transition t_2 of actor a_2.

Shown are the dependencies for the contraction of the transition sequence $\langle t_3, t_4 \rangle$ of actor a_3 into the transition t_c. One can observe that no additional de-pendency cycles have been introduced by this contraction.

Shown are the dependencies for the contraction of the transition sequence $\langle t_3, t_4 \rangle$ of actor a_2 into the transition t_d. Thus, inducing two (marked bold red) erroneous dependency cycles.

Fig. 3.12 Given (see (**a**)) is an example DFG containing two actors a_2 and a_3 used to show valid and invalid transition contractions. For this purpose, dependencies in transition sequences of both actors are shown in (**b**), (**c**), and (**d**). Here, *dashed lines* represent dependencies induced by the availability of data (tokens) and *solid lines* represent dependencies induced by the sequential nature of the FSM. Dependency cycles in these transition traces are marked by bolding the corresponding edges

an FSM transition is infeasible. Instead, a CSDF phase μ_j is matched by a transition sequence. To keep track of the number of tokens already produced and consumed for a CSDF phase, each FSM state q_n will be annotated with a tuple $\omega_n = (j, \text{cons}, \text{prod})$ where cons and prod are the vectors of number of tokens which are still to be consumed and produced to complete the consumption and production of the CSDF phase μ_j and j denotes the CSDF phase which should be matched.

The validation algorithm starts by deriving the tuple ω_0 (see Line 4) for the initial state q_0 from the first CSDF phase μ_0 of the *classification candidate* $\boldsymbol{\mu}$. The algorithm uses the set `ann` as a function $\omega_n = \texttt{ann}(q_n)$ from an FSM state to its

annotated tuple ω_n. Initially, the ann set is empty (see Line 6) denoting that all function values $\texttt{ann}(q_n)$ are undefined, and hence no tuples have been annotated. The annotation of tuples starts with the initial tuple $\omega_0 = \texttt{ann}(q_0)$ for the initial state q_0 by adding the corresponding association to the ann set in Line 8.

The algorithm proceeds by performing a *breadth-first search* (see Lines 5, 9, 12 and 27) of all states q_n of the FSM R starting from the initial state q_0 (see Line 7). For each visited state q_{src} (see Line 10) its annotated tuple ω_{src} will be derived from the set ann (see Line 11) and the corresponding outgoing transitions t of the state q_{src} (see Line 13) are checked against the annotated tuple ω_{src} (see Lines 14 to 19) via the transition criteria as given below:

Definition 6 (Transition Criterion I [45]). Each outgoing transition $t \in T_{\text{src}} = \{t \in T \mid t.q_{\text{src}} = q_{\text{src}}\}$ of a visited FSM state $q_{\text{src}} \in Q$ must consume and produce less or equal tokens than specified by the annotated tuple ω_{src}, i.e., $\forall t \in T_{\text{src}} : \omega_{\text{src}}.\text{cons} \geq \mathbf{cons}(t, I) \wedge \omega_{\text{src}}.\text{prod} \geq \mathbf{prod}(t, O)$. Otherwise, the annotated tuple ω_{src} is invalid and the *classification candidate* $\boldsymbol{\mu}$ will be rejected.

Definition 7 (Transition Criterion II [45]). Each outgoing transition $t \in T_{\text{src}} = \{t \in T \mid t.q_{\text{src}} = q_{\text{src}}\}$ of a visited FSM state $q_{\text{src}} \in Q$ must not produce tokens if there are still tokens to be consumed in that phase after the transition t has been taken, i.e., $\forall t \in T_{\text{src}} : \omega_{\text{src}}.\text{cons} = \mathbf{cons}(t, I) \vee \mathbf{prod}(t, O) = \mathbf{0}$. Otherwise, the annotated tuple ω_{src} is invalid and the *classification candidate* $\boldsymbol{\mu}$ will be rejected.

Transition criterion I ensures that a matched transition sequence consumes and produces exactly the number of tokens as specified by the CSDF phase μ of the *classification candidate*. *Transition criterion II* ensures that a transition sequence induces the same data dependencies as the CSDF phase μ of the *classification candidate*. If the transition does not conform to the above transition criteria, then the *classification candidate* is invalid and will be rejected (see Lines 15 and 18). Otherwise, that is conforming to both criteria, the matched transition sequence can be condensed via Definition 5 to the matched CSDF phase.

After the transition has been checked, the tuple ω_{dst} for annotation at the transition destination state $t.q_{\text{dst}}$ is computed in Line 20 according to the equation given below:

$$\begin{aligned}
\text{cons}_{\text{left}} &= \omega_{\text{src}}.\text{cons} - \mathbf{cons}(t, I) \\
\text{prod}_{\text{left}} &= \omega_{\text{src}}.\text{prod} - \mathbf{prod}(t, O) \\
j' &= (\omega_{\text{src}}.j + 1) \bmod \tau \\
\omega_{\text{dst}} &= \begin{cases} (\omega_{\text{src}}.j, \text{cons}_{\text{left}}, \text{prod}_{\text{left}}) & \text{if } \text{cons}_{\text{left}} \neq \mathbf{0} \wedge \text{prod}_{\text{left}} \neq \mathbf{0} \\ (j', \mu_{j'}.\text{cons}, \mu_{j'}.\text{prod}) & \text{otherwise} \end{cases}
\end{aligned} \tag{3.8}$$

In the above equation, $\text{cons}_{\text{left}}$ and $\text{prod}_{\text{left}}$ denote the consumption and production vectors remaining to match the CSDF phase μ_j. If these remaining consumption and production vectors are both the zero vector, then the matching of the CSDF phase μ_j

to a transition sequence has been completed. Hence, the tuple ω_{dst} will be computed to match the next CSDF phase $\mu_{(j+1) \bmod \tau}$. (The notation $n = l \bmod m$ for values $l \in \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}, m \in \mathbb{N} = \{1, 2, \ldots\}$ is used to denote the common residue $n \in \{0, 1, 2, \ldots m-1\}$ which is a non-negative integer smaller than m such that $l \equiv n \pmod{m}$.) Otherwise, the tuple ω_{dst} will use the updated remaining consumption and production vectors as given by $\text{cons}_{\text{left}}$ and $\text{prod}_{\text{left}}$.

Finally, if no tuple is annotated to the destination state $t.q_{\text{dst}}$, then the computed tuple ω_{dst} is annotated to the destination state in Line 26 and the destination state is appended to the queue for the *breadth-first search* in Line 27. Otherwise, an annotated tuple is already present for the destination state (see Line 21). If this annotated tuple ω'_{dst} and the computed tuple ω_{dst} are inconsistent, then the classification candidate will be rejected (see Lines 21 to 23).

3.6 Hardware/Software Codesign with SysteMoC

SysteMoC is the core language of the SystemCoDesigner [15, 25] codesign framework and HSCD flow, which is briefly outlined in Fig. 3.13. In Step 1, the application to be implemented is written in SysteMoC. During an extraction step (Step 2 in Fig. 3.13), the SysteMoC actors and channels as well as their connections are automatically extracted from the SysteMoC application. As a result, the *network graph* as defined in Sect. 3.3 can be used as input to the DSE of the SystemCoDesigner flow. As a second input to DSE, the possible variety of architectures is modeled by an *architecture graph* [4] that is specified by the designer in Step 3. The architecture graph is composed of *resources* (shaded orange) and edges connecting two resources that are able to communicate with each other. Within SystemCoDesigner, resources are usually modeled at a granularity of processors, hardware accelerators, communication buses, centralized switches (crossbars), decentralized switches (Networks on Chip), memories. In this context, edges of the architecture graph are interpreted as links. Finally, *mapping edges* (black) are specified within Step 3 by the designer for each actor (bordered blue) and each channel (shaded green) of the network graph. A mapping edge indicates the option to implement and execute an actor on the resource it points to. For a channel, the corresponding mapping edge represents the possibility to use the associated resource as buffer for messages sent over the channel. Both graphs, i.e., network graph and architecture graph, together with the mapping edges form a *specification graph* [4] that serves as input for DSE in Step 4.

To exemplify the concept of a specification graph, we use again the network graph implementing Newton's iterative square root algorithm introduced in Sect. 3.3 and also depicted in Fig. 3.13 as part of the specification graph. Due to the approximation part of the square root algorithm, actor a_3 requires floating point division. In contrast, actor a_2 requires floating point multiplication to check if the error bound is already satisfied. Considering the architecture graph, a Central Processing Unit (CPU) r_{CPU}, a dedicated hardware accelerator r_{HW} for actor a_3, a memory r_{MEM}, and two buses r_{P2P} and r_{BUS} can be identified. Let us assume that

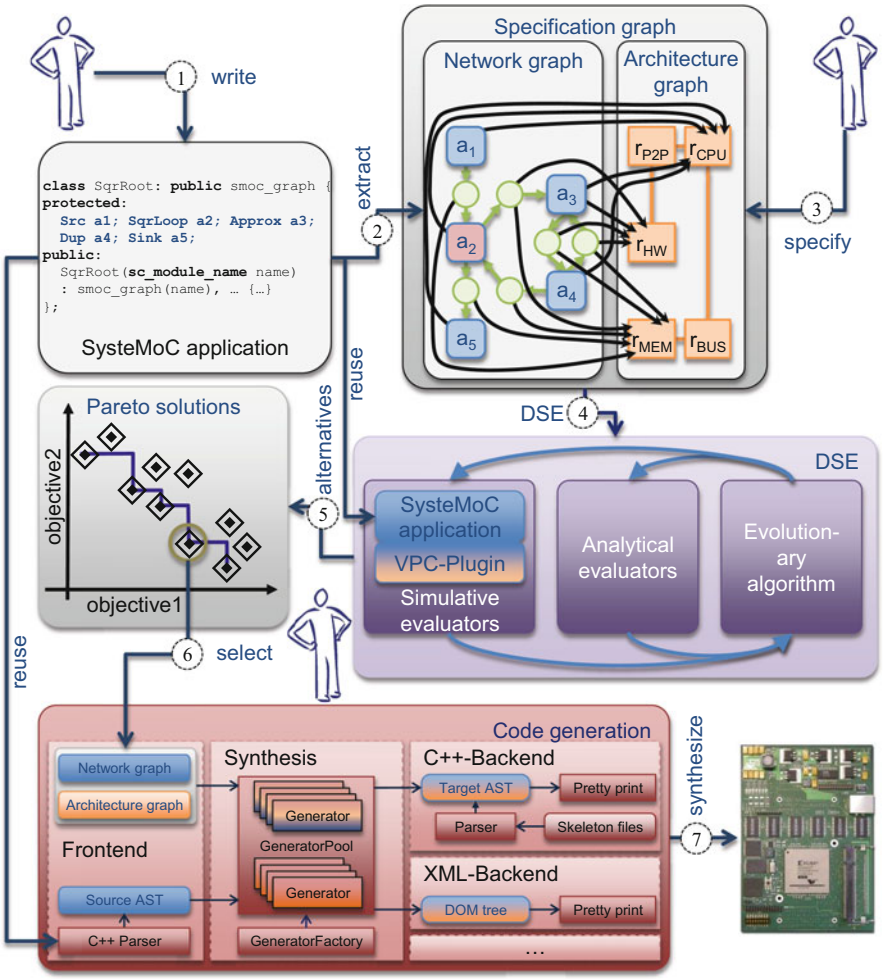
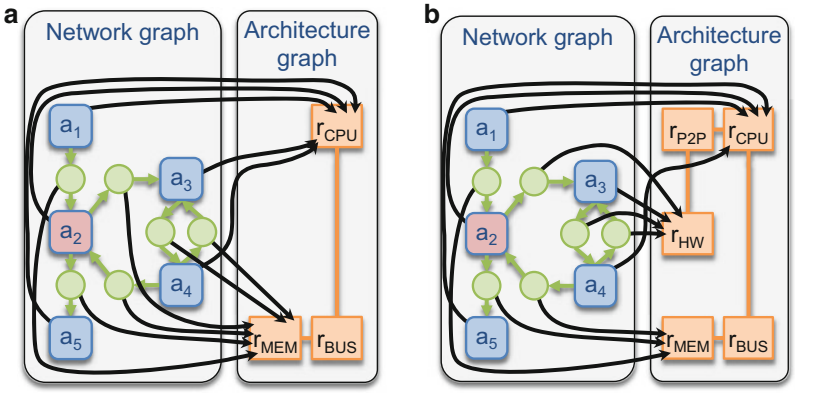


Fig. 3.13 Overview of a codesign flow using SysteMoC as input language for design space exploration of hardware/software alternatives as well as for subsequent code generation and hardware synthesis of each actor called SystemCoDesigner [15, 25]

r_{CPU} is a slow CPU that has no hardware support for floating point calculations. Hence, floating point calculations must be emulated in software. Thus, while all actors can be mapped to the CPU (see Fig. 3.14a), the actor a_3 will perform significantly worse on the CPU as compared to an implementation alternative (see Fig. 3.14b) where it is mapped to its dedicated hardware accelerator r_{HW} that is connected to the CPU via the point-to-point link r_{P2P}. Finally, the memory r_{MEM}, which is connected by the bus r_{BUS} to the CPU, is used to provide the program memory required by the CPU to implement the actors bound to the CPU as well as the buffer memory required by the channels of the network graph.



A mapping of actors using resource r_{CPU} that requires software emulation to support floating point operations

A mapping of actors using all available resources that have initially been specified in the architecture graph

Fig. 3.14 Depicted above are two possible hardware/software implementation alternatives for Newton's iterative square root algorithm. (**a**) A mapping of actors using resource r_{CPU} that requires software emulation to support floating point operations. (**b**) A mapping of actors using all available resources that have initially been specified in the architecture graph

As discussed above, the network graph of a specification can now be implemented in various ways (cf. Fig. 3.14). In DSE, different mappings of actors and communication channels to physical resources are explored. Resources of the architecture graph not being a target of a mapping are eliminated. As the mapping is independent of the actual MoC, any SysteMoC application can be explored during DSE. The DSE methodology tackles the problem of the exponential explosion of the number of ways the desired functionality can be implemented in an embedded system by performing an automatic optimization of the implementation of the functionality. For a more comprehensive overview of the DSE techniques supported within SystemCoDesigner, the interested reader is referred to ▸ Chap. 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration".

Moreover, the optimization needs a way to compare implementations with each other. Within SystemCoDesigner, a simulative performance analysis is performed for SysteMoC applications. For this purpose, a simulative evaluator [38] is realized by parameterizing [39, 43] a given SysteMoC application in such a way that it conforms to the design decisions taken by the DSE. This is established using run-time configurability via the VPC-Plugin [39] or MAESTRO-Plugin [36] in order to model the design decisions taken during DSE. As performance estimation is done by discrete-event simulation, only average case performance can be assessed during exploration for general SysteMoC models. Yet, if the explored network graph is substituted by less expressive models, e.g., by a static data-flow graph, a simple task graph, etc., then also formal model-based analysis techniques for timing, performance, and other objectives of interest such as reliability may be selected and added; see the box analytical evaluators in Fig. 3.13.

Finally, a SysteMoC application may serve as a golden model from which *virtual prototypes* for selected implementations can be derived as well via usage of a *synthesis back-end* (Step 7 in Fig. 3.13) for an implementation alternative selected by the designer (Step 6) from the output of the automatic optimization, i.e., the set of non-dominated or even Pareto-optimal implementations. During synthesis, actors are refined (compiled, synthesized) according to their binding. For this purpose, SystemCoDesigner currently supports hardware synthesis based on Cadence's Cynthesizer. The communication is refined with respect to the chosen model of communication, i.e., shared memory or message passing. So far [15], only shared memory communication using shared address spaces and point-to-point communications using hardware queue implementations are supported in the SystemCoDesigner synthesis back-end.

An important aspect of the SystemCoDesigner flow not shown in Fig. 3.13 is the exploitation of the analysis capability of SysteMoC applications. Although dynamic SysteMoC models can be automatically optimized and assessed by simulation, an additional optimization can be performed for static actors (cf. Sect. 3.5). SystemCoDesigner supports [37] the automatic *clustering* of subgraphs consisting of static actors. Clustering is a transformation where a subgraph of static actors is replaced by a single composite actor implementing a *quasi-static schedule* of the clustered static actors [17, 18]. Such a transformation can reduce the scheduling overhead resulting from dynamic scheduling significantly. For more information about clustering cf. [9].

3.7 Conclusions

Data-flow Models of Computation (MoCs) are well suited for the modeling of many applications that are targeted to heterogeneous hardware/software systems. Due to the inherently concurrent behavior of actors, partitioning into hardware and software blocks can be done at the granularity of actors. It is our recommendation therefore to think in terms of actors when designing a system because of the natural concurrency available in these models. However, there is a trade-off between expressiveness and analyzability of different data-flow MoCs. In this chapter, different data-flow models have been presented in a consistent framework. Moreover, SysteMoC, a MoC with a very high expressiveness, has been discussed. Its most outstanding property is the option to automatically identify if a SysteMoC actor uses this expressiveness or if it can be classified to belong to a more restricted data-flow MoC. In the latter case, this knowledge increases the analyzability. This advantage may be greatly exploited also during later optimization phases as well as during code generation and hardware synthesis from actor specifications to final hardware/software system implementations.

References

1. Bhattacharya B, Bhattacharyya SS (2001) Parameterized dataflow modeling for DSP systems. IEEE Trans Signal Process 49(10):2408–2421. doi:10.1109/78.950795
2. Bhattacharyya SS, Murthy PK, Lee EA (1997) APGAN and RPMC: complementary heuristics for translating DSP block diagrams into efficient software implementations. J Des Autom Embed Syst 2:33
3. Bilsen G, Engels M, Lauwereins R, Peperstraete J (1996) Cyclo-static dataflow. IEEE Trans Signal Process 44(2):397–408
4. Blickle T, Teich J, Thiele L (1998) System-level synthesis using evolutionary algorithms. Des Autom Embed Syst 3(1):23–58
5. Buck JT (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D dissertation, Department of EECS, University of California, Berkeley. Technical Report UCB/ERL 93/69
6. Commoner F, Holt AW, Even S, Pnueli A (1971) Marked directed graphs. J Comput Syst Sci 5(5):511–523. doi:10.1016/S0022-0000(71)80013-2
7. Dennis J (1974) First version of a data flow procedure language. In: Robinet B (ed) Programming symposium. Lecture notes in computer science, vol 19. Springer, Berlin/Heidelberg, pp 362–376. doi:10.1007/3-540-06859-7_145
8. Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y (2003) Taming heterogeneity – the ptolemy approach. Proc IEEE 91(1):127–144. doi:10.1109/JPROC.2002.805829
9. Falk J (2015) A clustering-based MPSoC design flow for data flow-oriented applications. Dr. Hut, Sternstr. 18, München. Dissertation Friedrich-Alexander-Universität Erlangen-Nürnberg
10. Falk J, Haubelt C, Teich J (2005) Syntax and execution behavior of SysteMoC. Technical report. 04-2005, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, D-91058 Erlangen
11. Falk J, Haubelt C, Teich J (2006) Efficient representation and simulation of model-based designs in systemC. In: Proceedings of the forum on specification and design languages (FDL 2006), pp 129–134
12. Falk J, Haubelt C, Zebelein C, Teich J (2013) Integrated modeling using finite state machines and dataflow graphs. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of signal processing systems. Springer, Berlin/Heidelberg, pp 975–1013
13. Falk J, Keinert J, Haubelt C, Teich J, Bhattacharyya SS (2008) A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In: Proceedings of the 8th ACM international conference on embedded software (EMSOFT 2008). ACM, New York, pp 189–198. doi:10.1145/1450058.1450084
14. Falk J, Schwarzer T, Glaß M, Teich J, Zebelein C, Haubelt C (2015) Quasi-static scheduling of data flow graphs in the presence of limited channel capacities. In: Proceedings of the 13th IEEE symposium on embedded systems for real-time multimedia (ESTIMEDIA 2015) p 10
15. Falk J, Schwarzer T, Zhang L, Glaß M, Teich J (2015) Automatic communication-driven virtual prototyping and design for networked embedded systems. Microprocess Microsyst 39(8):1012–1028. doi:10.1016/j.micpro.2015.08.008
16. Falk J, Zebelein C, Haubelt C, Teich J (2011) A rule-based static dataflow clustering algorithm for efficient embedded software synthesis. In: Proceedings of the design, automation and test in Europe (DATE 2011). IEEE, pp 521–526
17. Falk J, Zebelein C, Haubelt C, Teich J (2013) A rule-based quasi-static scheduling approach for static Islands in dynamic dataflow graphs. ACM Trans Embed Comput Syst 12(3):74:1–74:31. doi:10.1145/2442116.2442124

18. Falk J, Zebelein C, Keinert J, Haubelt C, Teich J, Bhattacharyya, SS (2011) Analysis of systemC actor networks for efficient synthesis. ACM Trans embed Comput Syst 10(2):18:1–18:34. doi:10.1145/1880050.1880054
19. Girault A, Lee B, Lee E (1999) Hierarchical finite state machines with multiple concurrency models. IEEE Trans Comput Aided Des Integr Circuits Syst 18(6):742–760
20. Gouda MG (1980) Liveness of marked graphs and communication and VLSI systems represented by them. Technical report, Austin
21. Gu R, Janneck JW, Raulet M, Bhattacharyya SS (2011) Exploiting statically schedulable regions in dataflow programs. Signal Processing Syst 63(1):129–142. doi:10.1007/s11265-009-0445-1
22. Hsu CJ, Bhattacharyya SS (2007) Cycle-breaking techniques for scheduling synchronous dataflow graphs. Technical report. UMIACS-TR-2007-12, Institute for Advanced Computer Studies, University of Maryland at College Park
23. Hylands C, Lee E, Liu J, Liu X, Neuendorffer S, Xiong Y, Zhao Y, Zheng H (2004) Overview of the ptolemy project, technical memorandum no. UCB/ERL M03/25. Technical report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
24. Kahn G (1974) The semantics of a simple language for parallel programming. In: IFIP Congress, pp 471–475
25. Keinert J, Streubühr M, Schlichter T, Falk J, Gladigau J, Haubelt C, Teich J, Meredith M (2009) SystemCoDesigner – an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. Trans Des Autom Electron Syst 14(1):1:1–1:23
26. Kosinski PR (1978) A straightforward denotational semantics for non-determinate data flow programs. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL 1978). ACM, New York, pp 214–221. doi:10.1145/512760.512783
27. Lee EA (2006) The problem with threads. Technical report. UCB/EECS-2006-1, EECS Department, University of California, Berkeley. The published version of this paper is in IEEE Computer 39(5):33–42, May 2006
28. Lee EA, Messerschmitt DG (1987) Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans Comput 36(1):24–35. doi:10.1109/TC.1987.5009446
29. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
30. Lee EA, Sangiovanni-Vincentelli AL (1998) A framework for comparing models of computation. IEEE Trans Comput Aided Des Integr Circuits Syst 17(12):1217–1229. doi:10.1109/43.736561
31. Parks TM (1995) Bounded scheduling of process networks. Ph.D dissertation, Department of EECS, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/2926.html. Technical Report UCB/ERL M95/105
32. Pino JL, Bhattacharyya SS, Lee E (1995) A hierarchical multiprocessor scheduling system for DSP applications. In: Proceedings of the Asilomar conference on signals, systems, and computers, vol 1, pp 122–126. doi:10.1109/ACSSC.1995.540525
33. Plishker W, Sane N, Kiemb M, Anand K, Bhattacharyya SS (2008) Functional DIF for rapid prototyping. In: The 19th IEEE/IFIP international symposium on rapid system prototyping (RSP 2008), pp 17–23. doi:10.1109/RSP.2008.32
34. Plishker W, Sane N, Kiemb M, Bhattacharyya SS (2008) Heterogeneous design in functional DIF. In: Proceedings of the 8th international workshop on embedded computer systems: architectures, modeling, and simulation (SAMOS 2008). Springer, Berlin/Heidelberg, pp 157–166. doi:10.1007/978-3-540-70550-5_18
35. Ptolemaeus C (ed) (2014) System design, modeling, and simulation using Ptolemy II. Ptolemy.org, Berkeley. http://ptolemy.org/systems
36. Rosales R, Glaß M, Teich J, Wang B, Xu Y, Hasholzner R (2014) Maestro – holistic actor-oriented modeling of nonfunctional properties and firmware behavior for mpsocs. ACM Trans Des Autom Electron Syst (TODAES) 19(3):23:1–23:26. doi:10.1145/2594481

37. Schwarzer T, Falk J, Glaß M, Teich J, Zebelein C, Haubelt C (2015) Throughput-optimizing compilation of dataflow applications for multi-cores using quasi-static scheduling. In: Stuijk S (ed) Proceedings of the 18th international workshop on software and compilers for embedded systems (SCOPES 2015). ACM, Berlin, pp 68–75
38. Streubühr M, Falk J, Haubelt C, Teich J, Dorsch R, Schlipf T (2006) Task-accurate performance modeling in systemC for real-time multi-processor architectures. In: Gielen GGE (ed) Proceedings of design, automation and test in Europe (DATE 2006). European Design and Automation Association, Leuven, pp 480–481. doi:10.1145/1131610
39. Streubühr M, Gladigau J, Haubelt C, Teich J (2010) Efficient approximately-timed performance modeling for architectural exploration of MPSoCs. In: Borrione D (ed) Advances in design methods from modeling languages for embedded systems and SoC's. Lecture notes in electrical engineering, vol 63. Springer, Berlin/Heidelberg, pp 59–72. doi:10.1007/978-90-481-9304-2_4
40. Stuijk S, Geilen M, Theelen BD, Basten T (2011) Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications. In: Proceedings of the international conference on embedded computer systems: architectures, modeling, and simulation (ICSAMOS 2011). IEEE Computer Society, pp 404–411. doi:10.1109/SAMOS.2011.6045491
41. Thiele L, Strehl K, Ziegenbein D, Ernst R, Teich J (1999) FunState – an internal design representation for codesign. In: White JK, Sentovich E (eds) ICCAD. IEEE, pp 558–565
42. Thiele L, Teich J, Naedele M, Strehl K, Ziegenbein D (1998) SCF – state machine controlled flow diagrams. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Gloriastrasse 35, CH-8092. Technical Report TIK-33
43. Xu Y, Rosales R, Wang B, Streubühr M, Hasholzner R, Haubelt C, Teich J (2012) A very fast and quasi-accurate power-state-based system-level power modeling methodology. In: Herkersdorf A, Römer K, Brinkschulte U (eds) Proceedings of the architecture of computing systems (ARCS 2012), vol 7179. Springer, Berlin/Heidelberg, pp 37–49. doi:10.1007/978-3-642-28293-5_4
44. Zebelein C (2014) A model-based approach for the specification and refinement of streaming applications. Ph.D. thesis, University of Rostock
45. Zebelein C, Falk J, Haubelt C, Teich J (2008) Classification of general data flow actors into known models of computation. In: Proceedings 6th ACM/IEEE international conference on formal methods and models for codesign (MEMOCODE 2008), pp 119–128

ForSyDe: System Design Using a Functional Language and Models of Computation

4

Ingo Sander, Axel Jantsch, and Seyed-Hosein Attarzadeh-Niaki

Abstract

The ForSyDe methodology aims to push system design to a higher level of abstraction by combining the functional programming paradigm with the theory of Models of Computation (MoCs). A key concept of ForSyDe is the use of higher-order functions as process constructors to create processes. This leads to well-defined and well-structured ForSyDe models and gives a solid base for formal analysis. The book chapter introduces the basic concepts of the ForSyDe modeling framework and presents libraries for several MoCs and MoC interfaces for the modeling of heterogeneous systems, including support for the modeling of run-time reconfigurable processes.

The formal nature of ForSyDe enables transformational design refinement using both semantic-preserving and nonsemantic-preserving design transformations. The chapter also introduces a general synthesis concept based on process constructors, which is exemplified by means of a hardware synthesis tool for synchronous ForSyDe models. Most examples in the chapter are modeled with the Haskell version of ForSyDe. However, to illustrate that ForSyDe is language-independent, the chapter also contains a short overview of SystemC-ForSyDe.

Acronyms

ASK	Amplitude Shift Key
CPS	Cyber-Physical System

I. Sander (✉)
KTH Royal Institute of Technology, Stockholm, Sweden
e-mail: ingo@kth.se

A. Jantsch
Vienna University of Technology, Vienna, Austria
e-mail: jantsch@ict.tuwien.ac.at

S.-H. Attarzadeh-Niaki
Shahid Beheshti University (SBU), Tehran, Iran
e-mail: h_attarzadeh@sbu.ac.ir

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_5

EDA	Electronic Design Automation
ForSyDe	Formal System Design
HSCD	Hardware/Software Codesign
MoC	Model of Computation
RTL	Register Transfer Level
SLD	System-Level Design

Contents

4.1 Introduction

Due to the ever increasing complexity of system-on-chip platforms and the continuous need for more powerful applications, industry has to cope with enormous challenges and faces exploding verification costs when designing state-of-the-art embedded systems. Still there are no systematic methods that can guarantee correct and efficient implementations at reasonable costs, in particular for systems that have to satisfy extra-functional properties like real-time behavior.

The problem is not new and well recognized. In 2007, Sangiovanni-Vincentelli discusses the problems of System-Level Design (SLD) [51] and states that "innovation in design tools has slowed down significantly as we approach a limit in the complexity of systems we can design today satisfying increasing constraints on time-to-market and correctness. The EDA community has not succeeded as of today in establishing a new layer of abstraction universally agreed upon that could provide productivity gains similar to the ones of the traditional design flow (Register Transfer Level (RTL) to GDSII) when it was first introduced."

The Formal System Design (ForSyDe) methodology addresses these challenges and aims at pushing the design entry to a considerably higher level of abstraction,

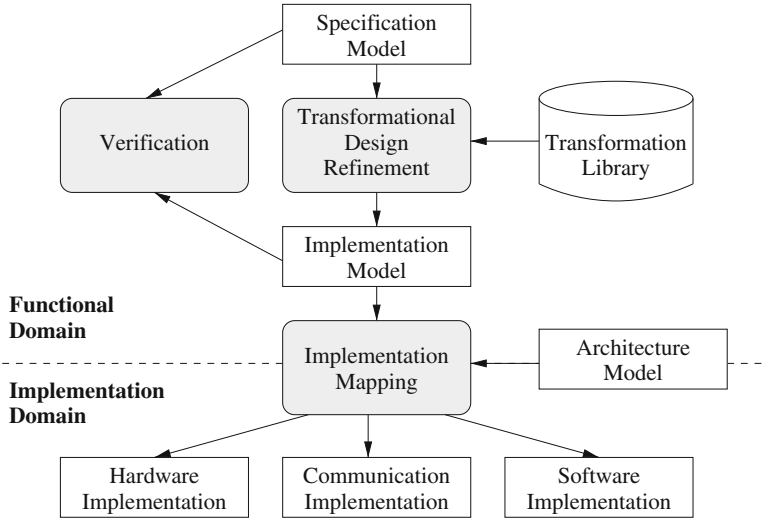


Fig. 4.1 ForSyDe system design flow [48]

by combining a formal base in form of the theory of Models of Computation (MoCs) [30] with an elegant system modeling technique based on the functional programming paradigm. This formal foundation enables the development of design transformation and synthesis techniques to convert the system model into the final implementation on a given target platform.

Figure 4.1 illustrates the ideas of the ForSyDe design flow as described in Sander's PhD thesis from 2003 [48]. The system design process starts with the development of an abstract, formal and functional *specification model* at a high abstraction level. The model is *formal* since it has a well-defined syntax and semantics. Furthermore, the model is based on well-defined models of computations, providing a clean mathematical formalism and an abstract communication model. It is *abstract* and *functional* since a system is modeled as a mathematical function of the input signals. This formal base of ForSyDe gives a good foundation for the integration of formal methods.

The synthesis process is divided into two phases. First, the specification model is refined into a more detailed *implementation model* by the stepwise *application of design transformations*. Since the specification model and implementation model are based on the same semantics, the same validation and verification techniques, i.e., simulation or formal verification, can be applied to both models. Design transformation is conducted in the *functional domain*. Inside the functional domain, a system model is expressed as a function using the semantics of ForSyDe. The second step in the synthesis phase is the mapping of the implementation model onto a given architecture. This phase comprises activities like partitioning, allocation of resources, and code generation. In the implementation mapping phase, the design process leaves the functional domain and enters the *implementation domain*, where

the design is described with "implementation-level languages," i.e., languages that efficiently express the details of the target architecture, such as synthesizable VHDL or Verilog for hardware and C for software running on a microcontroller. The task of the refinement process is to optimize the specification model and to add the necessary implementation details in order to allow for an efficient mapping of the implementation model onto the chosen architecture.

The objective of this book chapter is to give an overview of the current state of the ForSyDe design methodology, where special focus is given to the modeling framework. The whole chapter is written in a tutorial style, enabling the reader to experiment with the ForSyDe modeling and synthesis framework. Links to more detailed information are provided in the corresponding sections of the chapter. For readers not familiar with the functional programming language Haskell, a short overview of Haskell is provided in the appendix.

The chapter is structured as follows. Section 4.2 introduces the ForSyDe modeling framework and its key concepts, like signals, processes, process constructors, and MoC interfaces. Several ForSyDe MoCs are introduced and exemplified using the Haskell version of ForSyDe by concrete examples. Furthermore, the section presents the suitability of the functional paradigm to model reconfigurable processes, which in turn can be evolved to model adaptive systems. Finally, Sect. 4.2 concludes with a larger modeling case study consisting of several MoCs to illustrate the potential of the ForSyDe modeling framework. Section 4.3 introduces the ideas of transformational design refinement inside the functional domain and the use of the characteristic function to illustrate the consequences of semantic-preserving and nonsemantic-preserving design transformations to the designer. The synthesis of ForSyDe models to a target language is presented in Sect. 4.4. In particular, the section gives the general synthesis concepts that can be applied to any target architecture and exemplifies the general ideas by means of the ForSyDe hardware synthesis tool, which converts synchronous ForSyDe models to synthesizable VHDL. Section 4.5 illustrates the language independence of ForSyDe by introducing SystemC-ForSyDe, which implements the ForSyDe semantics in an industrial design language. Section 4.6 discusses related approaches, and finally Sect. 4.7 concludes the paper.

4.2 The ForSyDe Modeling Framework

In ForSyDe, a system is modeled as hierarchical concurrent process network. Processes communicate with each other only via signals. ForSyDe supports several Models of Computation (MoCs) and allows processes belonging to different models of computation to communicate via MoC interfaces as illustrated in Fig. 4.2. In order to formally describe the computational model of ForSyDe, the chapter uses a similar definition as the tagged signal model by Lee and Sangiovanni-Vincentelli [30].

The ForSyDe modeling elements are introduced and discussed in Sects. 4.2.1 and 4.2.2 using the synchronous model of computation and the Haskell implementation of the ForSyDe modeling framework, in short *Haskell-ForSyDe*. Section 4.2.3

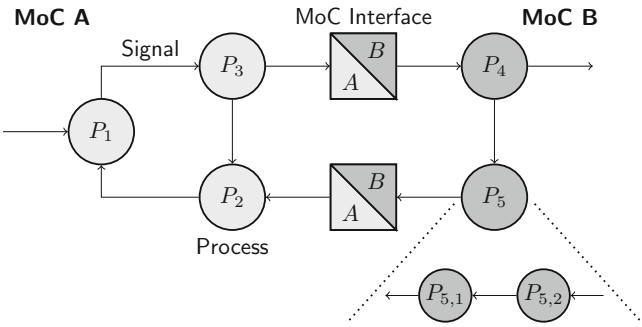


Fig. 4.2 A ForSyDe model is a hierarchical concurrent process network. Processes of different models of computation can communicate with each other via MoC interfaces. The process P_5 is created through process composition of the two processes $P_{5,1}$ and $P_{5,2}$

contains a deeper discussion of the synchronous MoC and also introduces additional ForSyDeMoCs. Heterogeneous ForSyDe models can be created by MoC interfaces, which are discussed in Sect. 4.2.4. Section 4.2.5 shows the usage of functions as signal values to model reconfigurable processes, and finally Sect. 4.2.6 concludes the discussion of the modeling framework with a larger modeling case study. All examples in this section have been modeled with the `forsyde-shallow` library, which is available on https://github.com/forsyde/forsyde-shallow, and have been run using version 7.10.3 of the Glasgow Haskell Compiler `ghc`.

It is important to point out that due to its pure functional paradigm, Haskell is a perfect match to the ForSyDe modeling framework. Still, the ForSyDe modeling formalism is language-independent and can be implemented in different languages. A good example is the SystemC implementation of ForSyDe, *SystemC-ForSyDe*, which is discussed in Sect. 4.5.

4.2.1 Signals

Processes communicate with each other by writing to and reading from signals. A signal is a sequence of *events*, where each event has a *tag* and a *value*. Tags can be used to model physical time, the order of events, and other key properties of the computational model. In the ForSyDe modeling framework, a signal is modeled as a list of events, where the tag of the event is either implicitly given by the event's position in the list as in the ForSyDe synchronous MoC or can be explicitly specified as in the case of the continuous-time or discrete-time MoC. The interpretation of tags is defined by the MoC. An identical tag of two events in different signals does not necessarily imply that these events happen at the same time. All events in a signal must have values of the same type. Signals are written as $\{e_0, e_1, e_2, \ldots\}$, where $e_i = (t_i, v_i)$ denotes the tag t_i and the value v_i of the i-th event in the signal.

In general, signals can be finite or infinite sequences of events and S is the set of all signals. The type of a signal with values of type D is denoted $S(D)$.

In order to distinguish ForSyDe signals from normal lists in Haskell, there is a special data type `Signal a` for signals carrying values of data type `a`. A signal of data type `a` is modeled as

```
data Signal a = NullS
          | a :- Signal a
```

and

```
s1 = 1:-2:-3:-4:-NullS
```

models a signal s_1 with integer values and has the data type `Signal Int`. The `Signal` data type is isomorphic to Haskell's list data type. The Haskell version of ForSyDe outputs signals in a more readable form, i.e., s_1 will be presented as `{1,2,3,4}`. The function `signal` can be used to convert a Haskell list into a ForSyDe signal, so another signal s_2 can be created by

```
s2 = signal [10,20,30,40]
```

resulting in the signal `{10,20,30,40}`.

The signals described so far have been finite signals, but infinite signals can be modeled in Haskell as well due to Haskell's lazy evaluation mechanism. The function `constS` creates an infinite signal of constant values.

```
constS x = x :- constS x
```

A full evaluation of the signal `constS 5` would not terminate. However, finite parts of infinite signals can be evaluated due to Haskell's call-by-need evaluation mechanism. The function `takeS` can be used for this purpose and returns the first n values of a signal, e.g., `takeS 3 (constS 5)` evaluates to `{5,5,5}`.

4.2.2 Processes

Processes are defined as functions on signals

$$p : S^m \rightarrow S^n = \underbrace{S \times S \times \cdots \times S}_{m} \rightarrow (\underbrace{S \times S \times \cdots \times S}_{n}).$$

The set of all processes is P.

Processes are functions in the sense that for a given set of input signals, always the same set of output signals is returned. Thus $s = s' \Rightarrow p(s) = p(s')$ is valid for a process with one input signal and one output signal. Note, that this still allows processes to have an internal state. A process does not necessarily react identical to the same event applied at different times. But it will produce the same, possibly infinite, output signal when confronted with identical, possibly infinite, input signals provided it starts with the same initial state.

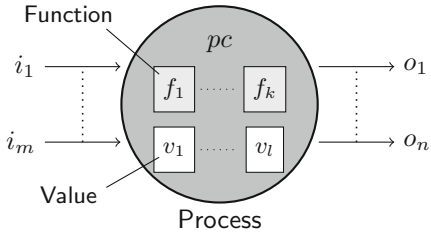


Fig. 4.3 A process is constructed by means of a process constructor *pc* that takes 0 to k side-effect-free functions and 0 to l values as argument

For processes with arbitrary number of input and output signals, the notation can become cumbersome to read. Hence, for the sake of simplicity, this chapter uses mostly processes with one input and one output only. This is not a lack of generality since it is straightforward to introduce *zip* and *unzip* processes which merge two input signals into one output signal and split one output signal into two output signals, respectively [27]. These processes together with appropriate process composition allow to express arbitrary behavior.

4.2.2.1 Process Constructors

Figure 4.3 illustrates the concept of *process constructor*, which is a key concept in ForSyDe originating from higher-order functions in functional programming languages. ForSyDe defines a set of well-defined process constructors, which are used to create processes. A process constructor *pc* takes zero or more side-effect-free functions $f_1, f_2, \ldots, f_k$ and zero or more values $v_1, v_2, \ldots, v_l$ as arguments and returns a process $p \in P$.

$$p = pc(f_1, f_2, \ldots, f_k, v_1, v_2, \ldots, v_l)$$

The functions represent the process behavior and have no notion of concurrency. They simply take arguments and produce results. The values model configuration parameters or the initial state of a process. The process constructor is responsible for establishing communication with other processes via signals. It defines the time representation, the communication interface, and the synchronization semantics. This separation of concerns leads to an elegant mathematical formalism that facilitates design analysis and design transformation. It is important to point out that most programming languages do not prevent the designer from creating functions that have side-effects, for instance by accessing a global variable inside a C-function. Since Haskell is a pure functional language, functions are side-effect free by design, but this property is not guaranteed in the SystemC version of ForSyDe (Sect. 4.5), where the designer has the responsibility not to use functions with side-effect.

A set of process constructors determines a particular MoC. The concept of process constructors ensures a systematic and clean separation of computation and communication. A function that defines the computation of a process can in principle be used to instantiate processes in different computational models. However, a computational model may impose constraints on functions. For instance,

the synchronous MoC requires a function to take exactly one event on each input and to produce exactly one event for each output (Sect. 4.2.3.1). Processes belonging to a data-flow MoC can consume and produce more than one token during each iteration, which has to be reflected in the computation functions for data-flow processes (Sect. 4.2.3.2).

The synchronous MoC is used to illustrate the usage of basic process constructors, which can be divided into combinational and sequential process constructors. A corresponding set of process constructors exists in the ForSyDe libraries for the other models of computation. Due to the total order of events in the synchronous MoC, the tag of an event is implicitly given by its position in the signal, so that synchronous ForSyDe signals do not carry an explicit tag.

A *combinational process constructor* creates combinational processes, i.e., processes that have no internal state. The basic combinational process constructor in the synchronous MoC is *mapSY*, which applies a function f to all signal values. Thus a process `twice` that doubles all input values of a synchronous signal `s` is modeled as

```
twice s = mapSY (*2) s
```

and can simulate the process *twice* with the input signal $s1$ as `twice s1`, which yields `{2,4,6,8}`. An adder can be modeled by

```
adder s1 s2 = zipWithSY (+) s1 s2
```

The process constructor *zipWithSY* applies a function f pairwise onto two synchronous signals. Hence, `adder s1 s2` yields `{11,22,33,44}` as output signal. The naming `mapSY`, `zipWithSY`, `zipWith3SY`, ... originates from functional programming, but to simplify for industrial designers the following aliases have been defined in ForSyDe for combinational process constructors: `combSY`, `comb2SY`, `comb3SY`, ...

A sequential process is a stateful process, where an output value depends not only on the current input values but also on the current state. The basic *sequential process constructor* is *delaySY*, which creates a process that delays a synchronous signal by one event cycle and where the current output value is given by the current state of the process. A register process can be modeled by

```
register s = delaySY 0 s
```

Here, the first argument to *delaySY*, in this case 0, is the initial state of the sequential process *delaySY* 0. Then `register s1` creates the output signal `{0,1,2,3,4}`, where 0, the initial state, is the value of the initial event. More powerful sequential processes and process constructors for finite state machines can be created by process composition as explained in the following section.

4.2.2.2 Process Composition

New processes can be created by composition of other processes to form a hierarchical process network. Figure 4.4 shows a model of a process *counter* that

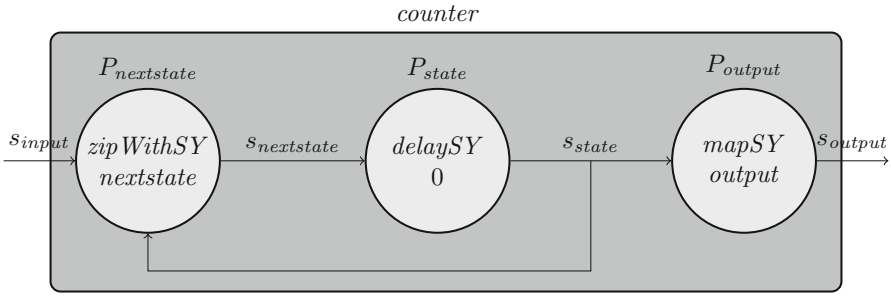


Fig. 4.4 The counter is modeled as concurrent process network. The process network is expressed as set of equations

is modeled with three processes: the combinational process $P_{nextstate}$ that calculates the next state, the sequential process P_{state} that holds the current state, and the combinational process P_{output} that models the output decoder. The counter shall output the value `TICK`, when the state is `0`, otherwise the event is absent. The possibility of the absence of an event is a special property of the synchronous MoC. It is modeled in ForSyDe by means of a special data type

```
data AbstExt a = Prst a
               | Abst
```

expressing that an event can either be present, and then has a value, or absent.

Figure 4.4 illustrates how more complex stateful sequential processes can be built by process composition around a basic sequential delay process that exists in all ForSyDe MoCs. It is worth to mention, that sequential processes in ForSyDe have a *local state*. However, due to its foundation in form of the theory of models of computation [30], where processes can only share information via signals, there is nothing like a shared global state in form of a shared variable. Since ForSyDe processes require side-effect-free functions as arguments, ForSyDe processes are deterministic in the sense that they will give the same output signals for the same history of input signals.

Zero-delay feedback loops in the synchronous MoC and related MoCs can cause causality problems, where a system specification might have no solution, a unique solution, or several solutions. ForSyDe deals with zero-delay feedback loops in a pragmatic way by simply forbidding it, following the same approach as the synchronous programming language Lustre [23]. More sophisticated solutions either calculate the least fix-point, as adopted by the synchronous languages Esterel [7] or Quartz [52], or using a relational approach, as adopted by the synchronous language Signal [12]. Another practical alternative is to introduce another level in the tag system, as adopted in VHDL, which includes micro steps in form of a delta-delay.

Listing 1 System model of counter in Haskell-ForSyDe

```
module Counter where

import ForSyDe.Shallow

data Direction = UP
               | HOLD deriving (Show)

data Clock = TICK deriving (Show)

-- Step 1: Specification of process network
counter s_input = s_output
   where s_output = p_output s_state
         s_state = p_state s_nextstate
         s_nextstate = p_nextstate s_state s_input

-- Step 2: Selection of process constructors
p_nextstate = zipWithSY nextstate
p_state = delaySY 0
p_output = mapSY output

-- Step 3: Specification of leaf functions
nextstate state HOLD = state
nextstate 4     UP   = 0
nextstate state UP   = state + 1

output 0 = Prst TICK
output _ = Abst
```

A top-down design of a system model in ForSyDe is conducted in three steps:

1. The designer sketches the process network including the selection of the MoC and the communication between the processes.
2. The designer selects suitable process constructors for all processes in the process network, alternatively expresses a high-level process by a composition of other processes. In Fig. 4.4, the process constructors *zipWithSY* and *mapSY* are used to form combinational processes, while the process constructor *delaySY* is used to model a process with internal state.
3. The designer formulates the arguments to the process constructors, i.e., the leaf functions (*nextstate*, *output*) and other parameters (initial state for *delaySY* is 0), to form ForSyDe processes.

Listing 1 shows the full ForSyDe model of the counter in Haskell-ForSyDe, including data types for the input (`Direction`) and output signals (`Clock`).

Haskell is a strongly typed language and although no data types are given, Haskell can infer the data type of the process `counter` to

```
counter :: Signal Direction -> Signal (AbstExt Clock)
```

This means that `counter` is a function that takes an input signal with data of type `Direction` and produces a signal with possibly absent events of type `Clock`. A simulation of the counter in ForSyDe shows the absent events as `'_'`-characters.

```
*Counter> counter (signal ([HOLD,UP,HOLD,UP,UP,UP,UP,HOLD,UP]))
{TICK,TICK,_,_,_,_,_,TICK,TICK,_}
```

4.2.3 ForSyDe Models of Computation

The Haskell-ForSyDe library supports several MoCs. The following subsections introduce the synchronous MoC (Sect. 4.2.3.1), two data-flow MoCs (Sect. 4.2.3.2), and finally the continuous-time MoC (Sect. 4.2.3.3). These three MoCs form a good base for the challenging design of Cyber-Physical Systems (CPSs), which integrate computation with physical processes [14]. The computation system consisting of software and digital hardware is naturally modeled with data-flow MoCs and synchronous MoC, while the physical process or plant is usually modeled with a continuous-time MoC. Section 4.2.6 presents a case study that integrates the presented MoCs of this section.

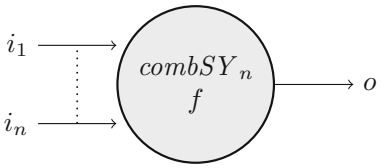
4.2.3.1 Synchronous Model of Computation (MoC)

The family of synchronous languages [5, 6], consisting of languages like Esterel [7], Lustre [23], Signal [12], or Quartz [52], is based on the synchronous MoC and uses the *perfect synchrony assumption*, i.e., *neither computation nor communication takes time*. Timing is entirely determined by the arriving of input events because the system processes input samples in zero time and then waits until the next input arrives. If the implementation of the system is fast enough to process all input before the next sample arrives, it will behave exactly as the specification model. ▸ Chapter 2, "Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems" contains a more detailed discussion about synchronous languages in general and the synchronous language Quartz in particular.

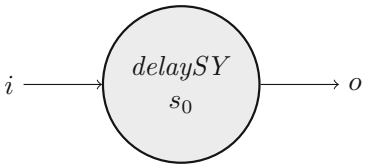
Synchronous processes are defined by the following specific characteristic. All synchronous processes consume and produce exactly one event on each input or output in each evaluation cycle, which implies a *total order* of all events in any signal inside a synchronous MoC. Events with the same tag appear at the same time instance. The set of synchronous processes is $P_{SY} \subset P$.

To model asynchronous or sporadic events like a reset signal, ForSyDe uses the special value $\perp$ to model the *absence* of an event. A value set V that is extended with the absent value $\perp$ is denoted $V_\perp = V \cup \{\perp\}$. It is often practical to abstract a non-absent value with the value $\top$. For convenience we call an event with an absent value an *absent event* and an event with a non-absent value a *present event*.

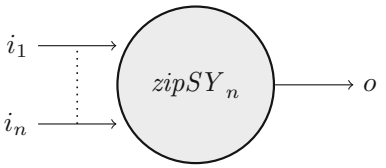
Figure 4.5 gives a formal definition of the set of basic process constructors and processes, which are needed to model a system in the synchronous MoC. In other models of computations, the set of basic process constructors is similar. Process constructors in the synchronous domain have the suffix "*SY*." Together with process composition, this set of combinational process constructors is sufficient to model systems inside the synchronous MoC.



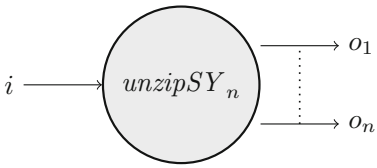
$$p = combSY_n(f) \in P_{SY}$$
where
$$\begin{aligned} o &= p(i_1, \ldots, i_n) \\ o[k] &= f(i_1[k], i_2[k], \ldots, i_n[k]) \end{aligned}$$



$$p = delaySY(s_0) \in P_{SY}$$
where
$$\begin{aligned} o &= p(i) \\ o[k] &= \begin{cases} s_0 & k = 0 \\ i[k-1] & k > 0 \end{cases} \end{aligned}$$



$$p = zipSY_n \in P_{SY}$$
where
$$\begin{aligned} o &= p(i_1, \ldots, i_n) \\ o[k] &= (i_1[k], i_2[k], \ldots, i_n[k]) \end{aligned}$$



$$p = unzipSY_n(f) \in P_{SY}$$
where
$$\begin{aligned} (o_1, \ldots, o_n) &= p(i) \\ i[k] &= (v_1[k], \ldots, v_n[k]) \\ o_j[k] &= v_j[k] \end{aligned}$$

Fig. 4.5 Formal definition of the basic process constructors *combSY*$_n$, *delaySY*, and the basic processes *zipSY* and *unzipSY*

A combinational process constructor *combSY*$_n$ takes a function $f : D_1 \times \cdots \times D_n \to E$ as argument and returns a process $p : S(D_1) \times \cdots \times S(D_n) \to S(E)$ with no internal state. The delay process constructor *delaySY* takes only one value $s_0 : D$ as argument and produces a process $p : S(D) \to S(D)$ that delays the input signal one cycle. The supplied value is the initial value of the output signal. The basic processes *zipSY*$_n$ and *unzipSY*$_n$ are required because a ForSyDe process is a mathematical function, which can only have a single signal as output. However, it is possible to model a process that has a signal of tuples as output and convert it with the process *unzipSY*$_n$ into a tuple of n signals. The process *zipSY*$_n$ converts a tuple of signals into a signal of tuples. Other process constructors are defined for convenience, such as the state machine constructor *mooreSY*$_n$, which is used to model a finite state machine, where the output depends only on the current state.

Figure 4.6 illustrates the process constructor *mooreSY*, which takes two functions, *ns* and *o*, and a value s_0 as arguments. The function *ns* calculates the next state, the function *o* calculates the output, and the value s_0 gives the initial state. Thus instead of specifying the counter of Fig. 4.4 with an explicit process network,

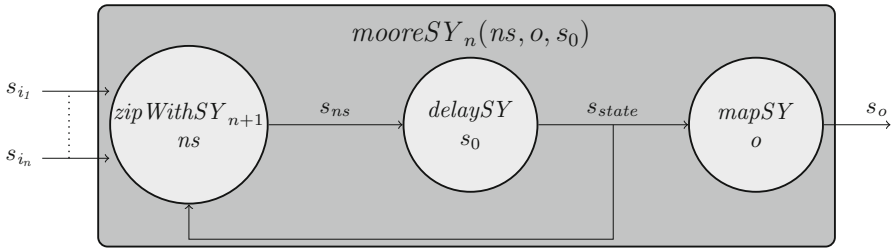


Fig. 4.6 The process constructor $mooreSY_n$ creates a synchronous process that models a state machine, where the output depends only on the state

the designer can use the process constructor *mooreSY* together with the arguments for *ns*, *o*, and s_0 to model the counter, i.e.,

```
counter = mooreSY nextstate output 0
```

This model has exactly the same behavior as the counter of Listing 1. The ForSyDe library also includes the *mealySY* process constructor to model synchronous Mealy FSMs, where the output depends not only on the state but on the input signal as well.

4.2.3.2 Data-Flow Models of Computation

ForSyDe provides several libraries for data-flow models of computation. These MoCs are untimed and there is no total order between events in two different signals, only a partial order exists. Many data-flow models exist in the literature ranging from models providing a high degree of expressiveness at the cost of low analyzability, like dynamic data flow, to models providing high analyzability at the cost of limited expressiveness, like synchronous data flow. ▸ Chapter 3, "SysteMoC: A Data-Flow Programming Language for Codesign" gives a very detailed overview about the most common data-flow Model of Computation (MoC) and introduces SysteMoC, a language to model and design data-flow systems based on SystemC.

This section will introduce two of ForSyDe's untimed data-flow MoCs: the *untimed MoC* provides a high level of expressiveness, while the *synchronous data flow (SDF) MoC* provides a high level of analyzability.

The ForSyDe SDF MoC follows the definition of synchronous data flow [29]. An SDF actor is created by process constructors, which take consumption rates, production rates, and a function as arguments and produce a process (actor) as result. Internally, an actor process $actorSDF_{m,n}$ with *m* input and *n* output signals is created by a composition of a $zipWithSDF_m$ process constructor and an $unzipSDF_n$ process.

Figure 4.7 illustrates the usage of the SDF MoC library by creating a system that repetitively takes two tokens from an input signal and compares them with the current maximum. The system model consists of two processes, which are created using an $actorSDF_{2,2}$ and a *delaySDF* process constructor. The process constructor $actorSDF_{2,2}$ takes three arguments. The first one, (2,1), gives the consumption rate for the input signals. The second one, (2,1), gives the production rate for the output

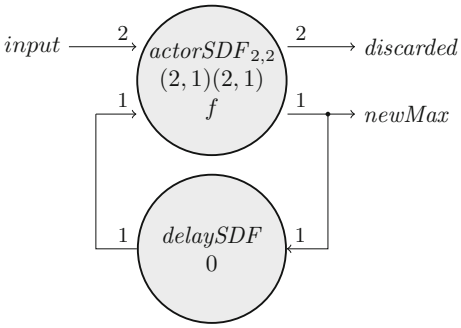


Fig. 4.7 Process network of an SDF model that calculates the current maximum and discards the other values

signals, and the third one f gives the computation function that operates on the input tokens. The initial value for the maximum is set to 0, given by the argument to the *delaySDF* process constructor. In each iteration the current maximum is compared with two new input values to determine a new maximum. The new maximum is fed back to the system, while the two other values are discarded.

Listing 2 An SDF model that calculates the current maximum and discards the other values

```
1 system input = (discarded, newMax)
2    where (discarded, newMax)
3                  = actor22SDF (2,1) (2,1) f input curMax
4          curMax = delaySDF 0 newMax
5
6 f [a, b] [c] = [(delete newMax [a,b,c], [newMax])]
7    where newMax = maximum [a,b,c]
```

The corresponding ForSyDe code is given in Listing 2. The function arguments in the SDF-MoC operate on lists and return lists as output. This can be seen in Line 6 of Listing 2, where the function f takes lists of different size as input and returns another list with tuples of lists as output values. The standard Haskell function `delete` removes an element from a list and outputs the list in reversed order. A simulation of the process network returns the expected result in the form of a tuple of signals. The first signal consists of the discarded values, with a changed order due to the reversed output of the function `delete`, while the second signal consists of the current maximum values.

```
*SDF> system (signal [1..10])
({1,0,3,2,5,4,7,6,9,8},{2,4,6,8,10})
```

The usage of the SDF MoC library ensures a well-defined and analyzable SDF model, where all processes behave according to the rules of the SDF-MoC.

In contrast to the SDF MoC, the untimed MoC of ForSyDe gives a high grade of expressiveness at the cost of losing analyzability. The reason is that

processes in the untimed MoC base the decision on how many tokens to be consumed and produced during an iteration on the current state of the process. This enables to model processes that vary the consumption and production rates during their run time. The expressiveness of the untimed MoC is best illustrated using the process constructor *mealyU*, which creates a state machine of Mealy type. The first argument is a function γ that operates on the state of the process and returns the number of tokens to be consumed in the next iteration. Listing 3 illustrates the use of the *mealyU* process constructor and the γ-function by a tutorial example.

Listing 3 An untimed model based on the process constructor *mealyU* consuming a varying number of input tokens in each iteration

```
1 system = mealyU gamma nextstate output 0
2     where gamma state = state + 1
3           nextstate state xs = length xs
4           output state xs = [state]
```

The initial state of the system is 0 given by the last argument of the *mealyU* process constructor (Line 1). Thus due to the γ-function (Line 2), a single token will be consumed in the first iteration. The next state is determined by the function *nextstate* (Line 3), which is the number of the consumed tokens during an iteration, i.e., one token in the first iteration. Thus the result of the γ-function will be 2 in the second iteration and consequently an increasing number of tokens is consumed in following iterations. The function *output* outputs the current state of the process (Line 4). The simulation below shows the expected results and stops when there are not enough tokens in the input signal.

```
*Untimed> system (signal [1..100])
{0,1,2,3,4,5,6,7,8,9,10,11,12}
```

4.2.3.3 Continuous Time Model of Computation

The time base, i.e., the tag, for the continuous-time MoC is given by the set of the positive real numbers, $t \in \mathbb{R}_+$, allowing to model physical time. To model continuous-time systems, ForSyDe exploits one key property of functional programming languages: functions are first-class citizens and can be treated as normal values. A continuous-time signal is defined as a set of sub-signals, where each sub-signal is defined by its time interval and the function that is executed during this time interval. A signal s_1 that has the constant value 1 during the time interval between 0 and 0.4 and the constant value -0.5 during the time interval between 0.4 and 1.0 is modeled as

```
s1 = signal [SubsigCT ((\t -> 1.0), (0,0.4)),
             SubsigCT ((\t -> -0.5), (0.4,1.0))]
```

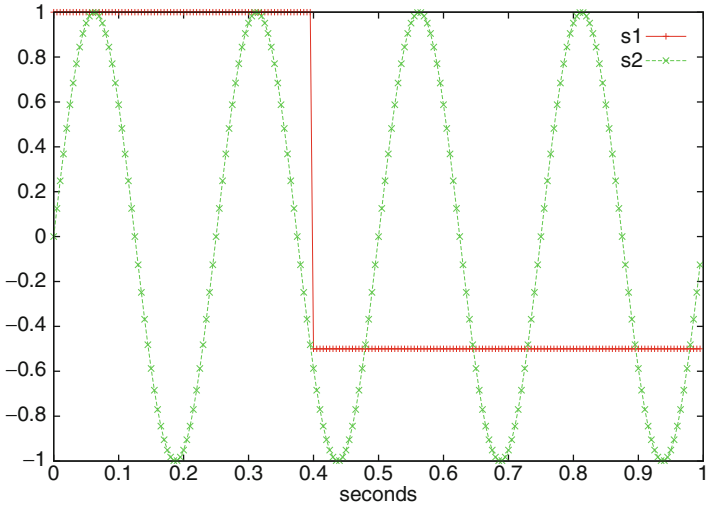


Fig. 4.8 The continuous-time signals s_1 and s_2 plotted (Resolution: 5 ms)

where `t-> 1.0` and `t-> -0.5` are the functions yielding a constant 1 or a constant 0.5, respectively. In a similar way, a continuous-time signal for a sine wave can be constructed, and the ForSyDe library allows to model sine waves, with the function `sineWave`, which takes the frequency in Hz of the sine wave as argument. The signal s_2 models a sine wave with the frequency 4 Hz during the time interval between 0 and 1.0.

```
s2 = sineWave 4 (0,1.0)
```

The signals can be plotted using the ForSyDe command `plotCT'` with the desired resolution as illustrated in Fig. 4.8. To generate the plots, `plotCT'` requires an installation of `gnuplot`.

```
plotCT' 5e-3 [(s1, "s1"), (s2, "s2")]
```

Please note that due to the lazy evaluation of Haskell, ForSyDe only calculates the results of the functions when needed, for instance to plot the graph with the given resolution. Otherwise, functions are treated as normal values.

The process constructors in the continuous-time MoC correspond to the process constructors in the synchronous, i.e., *mapCT*, *zipWithCT*, or *delayCT*. Processes are created and composed in the same way as in the synchronous MoC. The process p_1 that adds two continuous-time signals and the process p_2 that multiplies two signals are modeled as follows:

```
p1 = zipWithCT (+)
p2 = zipWithCT (*)
```

Figure 4.9 shows the plot of the operations `p1 s1 s2` and `p2 s1 s2`.

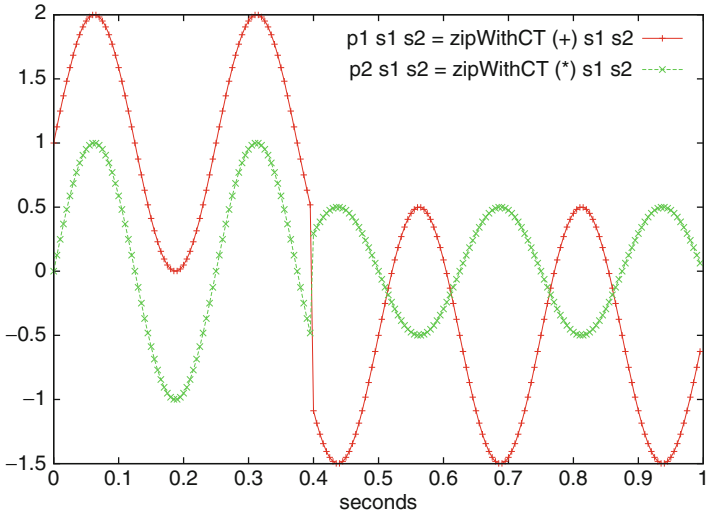


Fig. 4.9 Operations on the continuous-time signals s_1 and s_2 (Resolution: 5 ms)

4.2.4 Model of Computation Interfaces

ForSyDe supports heterogeneity through MoC interfaces, which are special types of processes used to connect signals belonging to different MoCs. Classical examples for practical MoC interfaces are analog-to-digital and digital-to-analog converters. Corresponding MoC interfaces also exist in ForSyDe for the connection of the continuous-time MoC and the synchronous MoC, in the form of `ct2sy` and `sy2ct`. Both interfaces are inspired by practical A/D and D/A converters.

The MoC interface `ct2sy` corresponds to an A/D converter and is an ideal process in the sense that is does not perform quantization of the input signal; it only samples the input signals according to the given sample period. To model a real A/D converter, an additional synchronous ForSyDe process `quantizer` is required that takes the minimal and maximal signal values and the number of bits as input and produces a quantized signal. There are two modes for the MoC interface `sy2ct`, which corresponds to a D/A converter. In `DAhold`-mode, the continuous-time output follows directly the synchronous input value for the whole sampling period, while in `DAlinear`-mode, a smooth transition between two adjacent synchronous values is done.

Listing 4 and the plot of the output signals in Fig. 4.10 illustrate the use of the MoC interfaces between the continuous-time MoC and synchronous MoC.

The example shows the effects of both an ideal A/D converter `adc_ideal` on the output signal s_5 and a nonideal A/D converter `adc_non_ideal` with a quantization stage by means of the output signal s_6. ForSyDe provides a few standard MoC interfaces but allows designers to write their own MoC interfaces. These interfaces can be on all abstraction levels and might be ideal, as in the case

Listing 4 Illustration of the use of MoC interfaces by means of the `ct2sy` and `sy2ct`, which are used to model ideal and nonideal A/D converters and a D/A converter. The output signals are plotted in Fig. 4.10

```
1  -- Ideal A/D-converter
2  adc_ideal = ct2sy 0.02
3  -- Non-ideal A/D-converter with quantizer
4  quantizer = mapSY (quantize (-1.0, 1.0) 4)
5  adc_non_ideal = quantizer . (ct2sy 0.02)
6  -- D/A-converter using DAhold-mode
7  dac = sy2ct DAhold 0.02
8
9  -- Sine wave as input signals
10 s4 = sineWave 1 (0,1.0)
11
12 -- Output signals : s6 is inverted for illustration purposes
13 s5 = (dac . adc_ideal) s4
14 negator = mapSY (* (-1.0))
15 s6 = (dac . negator . adc_non_ideal) s4
```

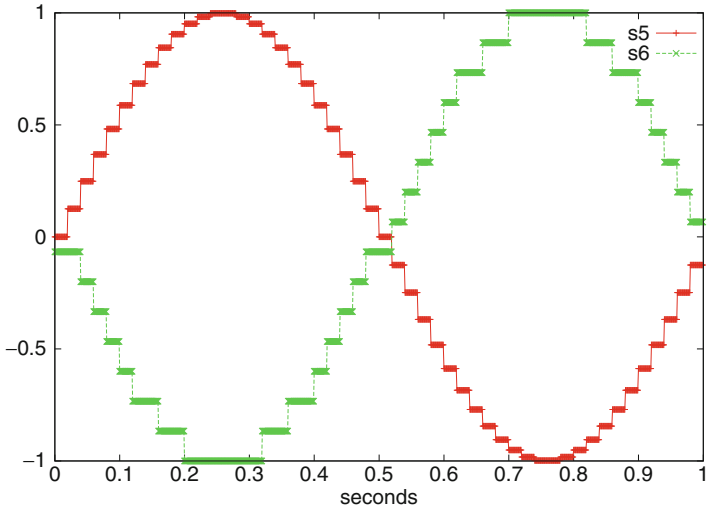
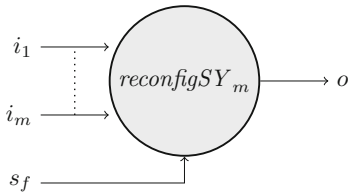


Fig. 4.10 The signal s_5 shows the effect of the sampling period on the input sine wave, but does not use any bit-level quantization. The signal s_6 shows also the effect of quantization with a resolution of 4 bits on a negated signal

for the standard `ct2sy` MoC interface, or can be nonideal as in the case of the `adc_non_ideal`, which in itself is a MoC interface created by the composition of `ct2sy` and the synchronous quantizer.



$$
\begin{aligned}
p_f &= reconfigSY_m \\
\text{where} & \\
o &= p_f(s_f, (i_1, \ldots, i_m)) \\
o[k] &= s_f[k](i_1[k], i_2[k], \ldots, i_m[k])
\end{aligned}
$$

Fig. 4.11 The process $reconfigSY_m$ models a reconfigurable process, where the function that the process executes is controlled by an input signal carrying functions

4.2.5 Reconfigurable Processes

The property of functional languages, that functions are regarded as first-class citizens, has already been used for the continuous-time MoC library. ForSyDe exploits this key property also to model *reconfigurable processes*, i.e., processes that change their behavior over time, by introducing signals carrying functions as event values. An example is the synchronous signal s_f, which has functions on numbers as signal values.

$$s_f = \{(+), (-), (*), (+)\}$$

Figure 4.11 illustrates how a signal $s_f : S(D_1 \times \cdots \times D_m \to E)$, where the values of the signal are functions, serves as input signal for a reconfigurable process. The reconfigurable process $p_f : S(D_1) \times \cdots \times S(D_m) \times S(D_1 \times \cdots \times D_m \to E) \to S(E)$ executes always the current value, i.e., a function, of the signal s_f. This means that the reconfigurable process does not need to provide the code for different modes of functions, because they are supplied from the outside. Reconfigurable processes can be implemented by run-time reconfigurable hardware or software, where the new functions can be loaded into a reconfigurable area, such as an FPGA or memory block, during operation.

Using the classification introduced by McKinley in [36], reconfigurable ForSyDe processes belong to the category of *compositional adaptation*. In contrast, most modeling frameworks offer only *parameter adaptation*, where adaptivity is changed by parameter settings or the existence of different system modes usually implemented by *if-then-else* statements.

Reconfigurable processes can be used to create a *self-adaptive process*, as illustrated in Fig. 4.12, where the executed function of the process is triggered by the change of the values of the input or output signals. The self-adaptive process p_{sa} is constructed as a process network consisting of a reconfigurable $p_{reconfig}$ and another process $p_{control}$ that controls the functionality of the reconfigurable process $p_{reconfig}$. At the highest level of abstraction, we assume adaptation to be instantaneous. Thus the change of functionality indicated by a new value of the signal s occurs at the same instant as the input or output values that trigger the change of the functionality of the adaptive process.

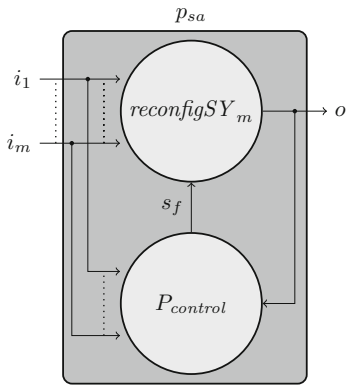


Fig. 4.12 A self-adaptive process p_{sa} is modeled as a process network of an adaptive process and an additional process. The signal s_f carries functions as values as illustrated in Fig. 4.11

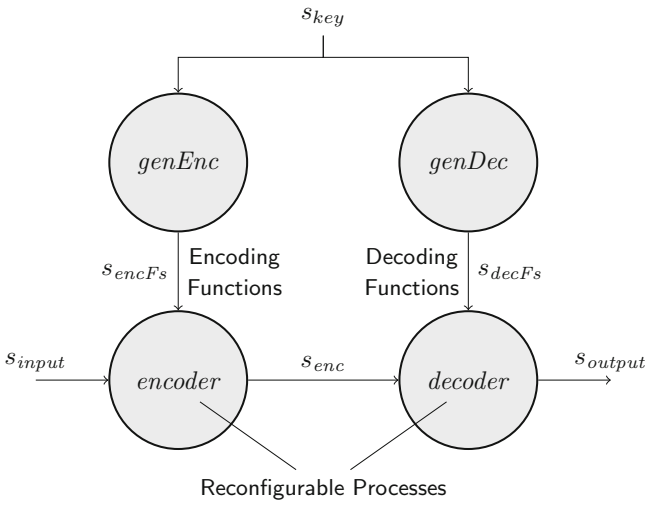


Fig. 4.13 Encoder-decoder

In order to show that reconfigurability can be treated as a first-class citizen in ForSyDe, we illustrate how the existing synchronous ForSyDe process constructor *combSY*$_n$ in conjunction with the function application operator (\$) can be used to model a synchronous reconfigurable process *reconfigSY*$_n$. The function application operator is defined as `f $ x = f x` and enables the application of functions on signal values.

```
reconfigSY = combSY ($)
reconfig2SY = comb3SY ($)
```

Figure 4.13 shows a tutorial example to model run-time reconfigurability using a synchronous system model with two reconfigurable processes. A signal is

encoded with an encoding function and later the encoded signal is decoded with a decoding function. The signal s_{key} is an input to both the *genEnc* and *genDec* processes. The processes *encoder* and *decoder* are reconfigurable processes and have signals carrying functions as inputs. Figure 4.13 models reconfigurability at a high abstraction level, where the reconfiguration of the reconfigurable process is assumed to be instantaneous and does not consume any time. The corresponding ForSyDe source code is given in Listing 5.

Listing 5 ForSyDe source code for the encoder-decoder example

```
1  module EncoderDecoder where
2
3  import ForSyDe.Shallow
4
5  reconfigSY fs xs = zipWithSY ($) fs xs
6
7  genEnc s_key = mapSY f s_key
8                 where f x y = y + x
9
10 genDec s_key = mapSY f s_key
11                where f x y = y - x
12
13 encoder s_encFs xs = reconfigSY s_encFs xs
14
15 decoder s_decFs xs = reconfigSY s_decFs xs
16
17 system s_key s_input  = (s_enc, s_output)
18        where s_output = decoder s_decFs s_enc
19              s_enc    = encoder s_encFs s_input
20              s_encFs  = genEnc s_key
21              s_decFs  = genDec s_key\vspace*{5pt}
```

The simulation with the signal `s_key = signal [1,4,6,1,1]`, which carries the encoding keys, and an input signal `s_input = signal [1,2,3,4,5]` yields the expected output

```
*EncoderDecoder> system s_key s_input
({2,6,9,5,6},{1,2,3,4,5})
```

where the output is a tuple of two signals. The first signal `{2,6,9,5,6}` is the encoded signal s_{enc} and the second signal `{1,2,3,4,5}` is the decoded output signal s_{output}.

The processes *encoder* and *decoder* in Fig. 4.13 can be further refined in consecutive design steps to take reconfiguration time and the need for buffers into account. This would reflect the nature of partial and run-time reconfigurable FPGAs. For a more detailed discussion about the modeling and refinement of reconfigurable and adaptive systems in ForSyDe, see [50].

4.2.6 Modeling Case Study

To illustrate the capability of the ForSyDe modeling approach, a case study from the European FP6 ANDRES project [25] will be used. The case study models an Amplitude Shift Key (ASK) transceiver and combines three different MoCs: the continuous-time MoC, the synchronous MoC, and the untimed MoC.

The structure of the system is illustrated in Fig. 4.14. A synchronous signal of integers enters the system in (1) and is converted to a signal of bit vectors, which are then encoded. The encoded signal is converted into a serial bit stream, before it is converted into a continuous-time signal in (2). This signal is then modulated using an amplitude shift key modulation and amplified before it leaves the sender of the transceiver in (3). In order to test the system, a Gaussian noise (4) is added to the signal resulting in a noisy signal in (5). This signal is received by the transceiver and is converted from the continuous-time MoC to the untimed MoC. Then the serialized signal is converted into a signal of bit vectors (6), before it is decoded and converted to an output signal of integers (7).

The system has an inbuilt mechanism to deal with noise during transmission. In case bit errors are detected after (6), the adaptive power controller (8) increases the gain and the amplification of the transmitted signal will be increased in the process *adaptGain* (9).

The operation of the system is visible in the simulation in Fig. 4.15. At the start of the simulation, all input signals are either fully available to the simulator as in the case of the synchronous input signal (1) or are defined as source processes, like in the case of the Gaussian noise generator (4), which can produce an infinite signal thanks to Haskell's lazy evaluation mechanism. The simulation is data-driven and ends when the final event in the synchronous input signal (1) has been processed. The model contains several MoC interfaces, which define the relation between the tag systems of the different MoCs as discussed in Sect. 4.2.4. The synchronous input signal (1) is represented as a signal of integers. The Gaussian noise increases between 2 and 3 ms (4). This causes a bit error in the third event of the output signal (7). The error is detected and causes to amplify the encoded signal to be transmitted in the following event cycle from 3 to 4 ms (3). The following event is received correctly, which can also be seen from the noisy ASK signal in (5) and the amplification power is lowered again to normal level.

4.3 Transformational Design Refinement

The ForSyDe design process starts with the development of an initial abstract *specification model* that defines the behavior of the system as a function between system inputs and system outputs based on the tagged signal model [30]. A central idea of ForSyDe is to exploit the formal nature of this functional system model for design transformation and to refine the *specification model* by the application of well-defined design transformations into a lower-level *implementation model*, which

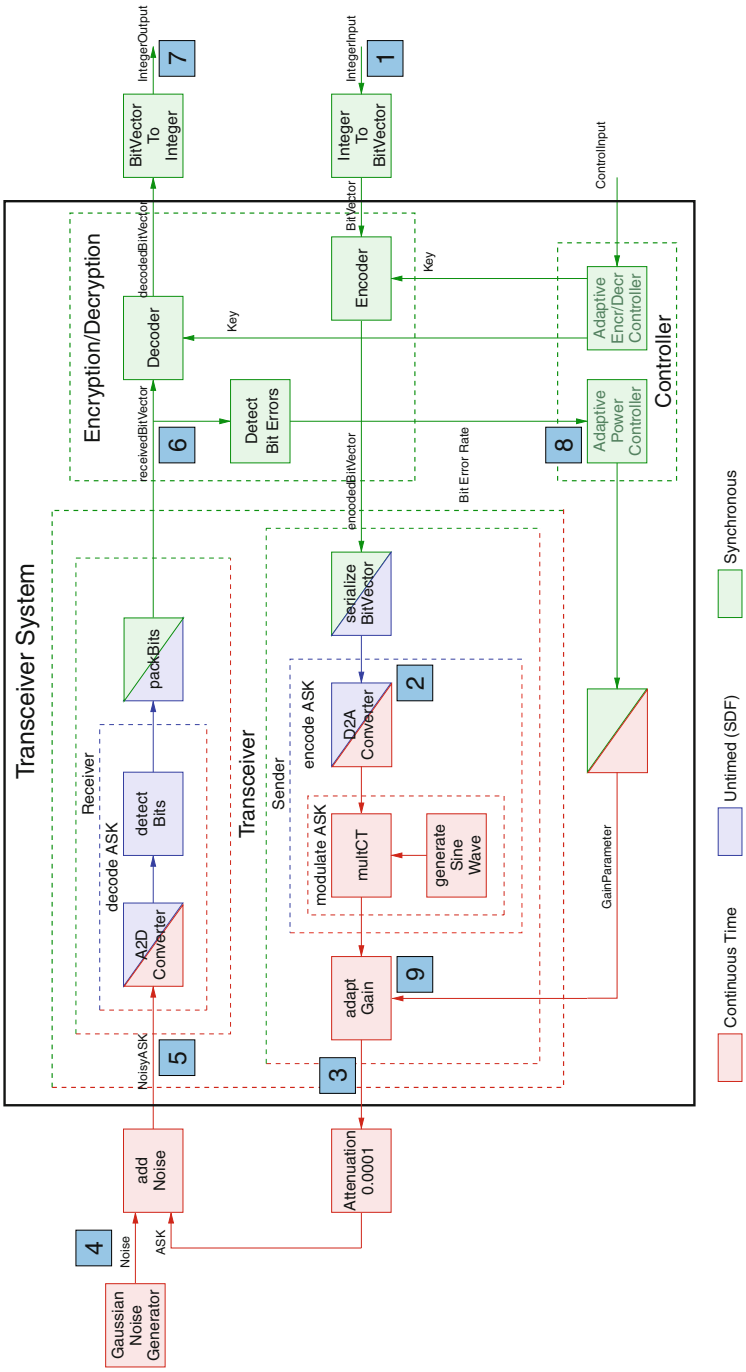


Fig. 4.14 Structure of the ASK transceiver example

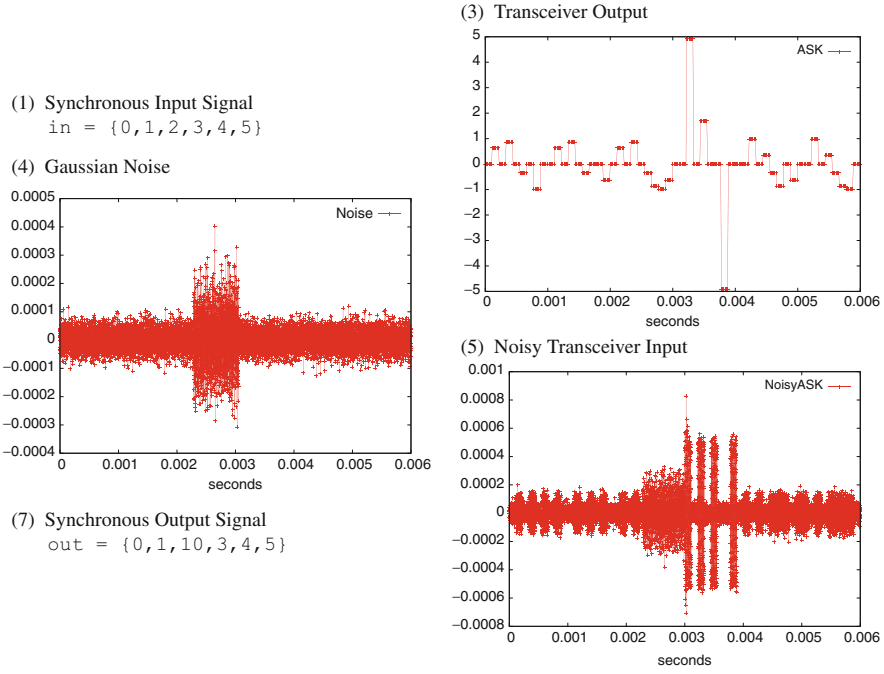


Fig. 4.15 Simulation of the ASK transceiver case study

is then synthesized into the target platform (Sect. 4.4). Since both the specification and the implementation model are based on the same ForSyDe semantics, as described in Sect. 4.2, they can be simulated using the same testbench as long as the specification of the inputs and outputs do not change during the refinement process.

The transformational design refinement process requires both *semantic-preserving* transformations, which do not change the meaning, i.e., the timely and functional behavior, of the model, and *nonsemantic-preserving* transformations or *design decisions*, which change the meaning of the model. Nonsemantic-preserving design decisions are required to improve the efficiency of the model, for instance, to refine an infinite buffer into a finite buffer or for data type refinement, but require an additional verification effort.

In order to give the designer information about the implications on the behavior of a refined process due to the application of a design transformation rule, ForSyDe has introduced the concept of *characteristic function* and exemplified it for the synchronous MoC [49]. The characteristic function F_{PN} of a process network PN, where

$$PN(i_1, \ldots, i_m) = (o_1, \ldots, o_n)$$

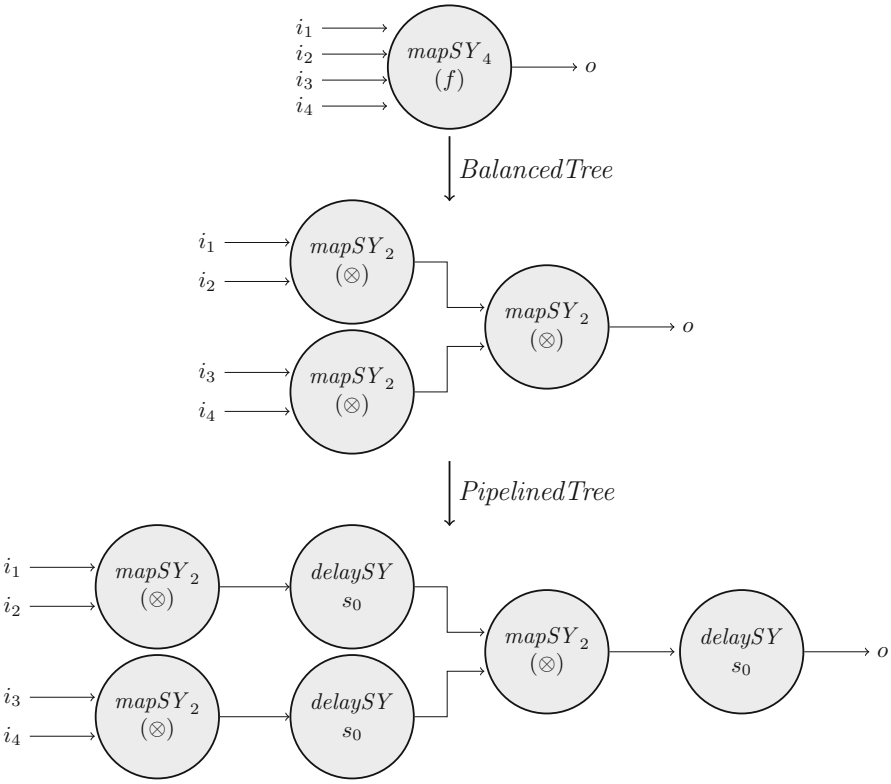


Fig. 4.16 Transformation of a combinational function into a balanced, pipelined tree structure ($m = 4$)

expresses the functional behavior of a process network as the dependence of the output events at tag j on the input signals, i.e.,

$$F_{PN}(i_1, \ldots, i_m, j) = ((T(o_1[j]), V(o_1[j]), \ldots, (T(o_n[j]), V(o_n[j])))$$

where $T(e)$ denotes the tag of the event e, $V(e)$ denotes the value of the event e, and $s[j]$ denotes the j-th event of a signal s. Thus $T(o_i[j])$ and $V(o_i[j])$ give the tag and the value of the j-th event $o_i[j]$ of the output signal o_i. The characteristic function utilizes the property of the tagged signal model that divides an event into tag and value. This enables to compare the behavior of two different process networks with respect to both timing (tag) and computation function (value).

Figure 4.16 illustrates transformational design refinement using the semantic-preserving transformation rule *BalancedTree* and the design decision *PipelinedTree* in order to convert a process network *PN* performing the mathematical function $f(x_1, x_2, \ldots, x_m) = x_1 \otimes x_2 \otimes \cdots \otimes x_m$, where $m = 2^k; k \in \mathbb{N}_+$, into a pipelined

representation PN' that requires two input functions. The characteristic function informs the designer about the implication of the design transformation, which in this case is a delay of the output compared to the original process network PN by k event cycles, i.e.,

$$F_{PN'}(i_1, \ldots, i_m, j) = F_{delaySY_k(s_0) \circ PN}(i_1, \ldots, i_m, j); \quad \forall j \geq k$$

In other words, after this transformation, the refined process network will yield the same output values as the original process network after a delay of k event cycles. Thus during transformational design refinement, the characteristic function is used to inform the designer on both changes in the timely behavior and changes of the output values for a given design transformation rule.

The implication can be used by the designer to verify the local consequences of the design transformation. However, a transformation like *PipelinedTree* also affects the global timing of a larger process network and has to be compensated in feedback loops or parallel paths in the global process network. This section can only give an overview about transformational design in ForSyDe. For a more detailed discussion, see [49] as main reference but also [45], which focuses mainly on *nonsemantic-preserving transformations* and discusses the verification of design decisions at the local level and gives a method for restoring time correctness at the global level. So far only the concepts for design transformations have been proposed [45, 49], the automation of the design transformation is still a topic for future work.

4.4 Synthesis of ForSyDe Models

Thanks to the well-defined structure of ForSyDe models, it is possible to give a general scheme for the synthesis of a ForSyDe implementation model into an implementation on a given target platform. The general synthesis flow is described in Sect. 4.4.1 and then exemplified by hardware synthesis of ForSyDe models belonging to the synchronous MoC in Sect. 4.4.2. The general synthesis concepts have also been applied for software synthesis toward a single processor as part of a case study on Hardware/Software Codesign (HSCD) [32]. Furthermore, there exists a first version of a synthesis tool `f2cc` targeting GPGPUs from abstract ForSyDe specifications where the functions are expressed in C-code [11].

Please note that this section deals with the translation of ForSyDe models into the target implementation. The refinement of the specification model through design transformations into the implementation model has been the topic of Sect. 4.3.

4.4.1 General ForSyDe Synthesis Concepts

ForSyDe models are structured as hierarchical concurrent process networks, where each process is either (1) composed of other processes communicating via signals, is (2) constructed by means of a process constructor, or is (3) a basic process. In order

to synthesize a system model into a target implementation, synthesis rules have to be developed for these different cases.

1. **Synthesis of concurrent process networks.** ForSyDe process networks communicate via signals according to a well-defined model of computation. The process network needs to be translated into a corresponding implementation in the target language that obeys the properties of the model of computation.
2. **Synthesis of processes created by process constructors.** Each process constructor has to be implemented into the corresponding pattern in the target implementation. As a second step, the arguments of the process constructors, i.e., pure functions and values, have to be translated into the target implementation.
3. **Synthesis of basic processes.** Basic processes like *zipSY* and *unzipSY* need to be implemented in the target implementation.

4.4.2 Hardware Synthesis

The general synthesis concept is illustrated by means of the synthesis of ForSyDe system models belonging to the synchronous MoC into digital hardware, more precisely into the corresponding VHDL code. The translation of synchronous ForSyDe models is straightforward. The synchronous MoC can be faithfully implemented in synchronous hardware, where the total order of events is preserved by the use of hardware clocks. Furthermore, both ForSyDe system models and VHDL models are based on concurrent processes communicating via signals.

Figure 4.17 shows how ForSyDe processes are translated to VHDL components, while ForSyDe signals are translated to VHDL signals.

In order to synthesize ForSyDe processes based on process constructors to the corresponding hardware, the corresponding pattern for each process constructor has to be identified in VHDL, and the arguments of the process constructors, pure functions and variables have to be translated to synthesizable VHDL. Figure 4.18 shows the translation of the ForSyDe process constructors *mapSY*, *delaySY*, and

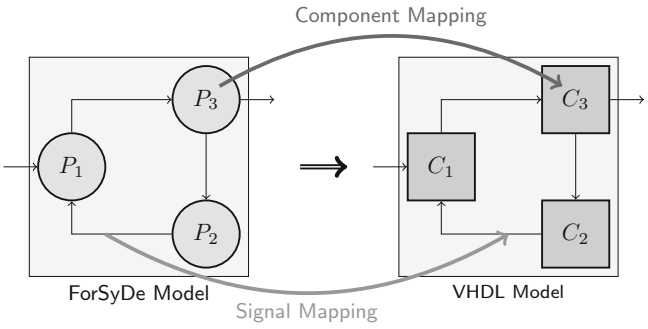


Fig. 4.17 Hardware synthesis of process networks

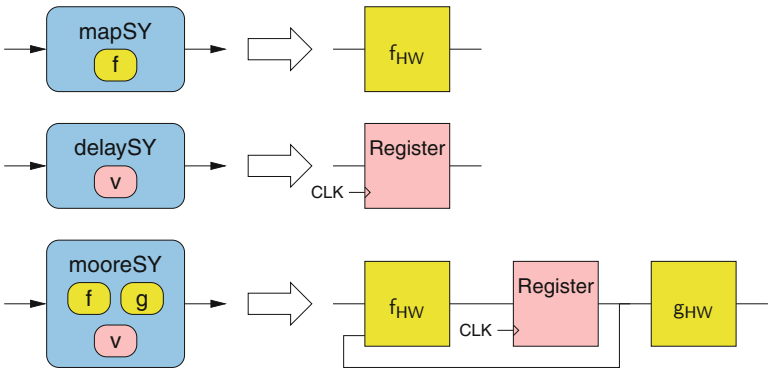


Fig. 4.18 Hardware synthesis of process constructors

mooreSY into the corresponding hardware patterns. Finally, the side-effect-free ForSyDe functions, which are the arguments of the process constructors, are translated into the corresponding VHDL functions.

4.4.3 ForSyDe Hardware Synthesis Tool

Based on the concepts presented in the previous sections, a hardware synthesis back end has been developed for ForSyDe [1]. The hardware synthesis tool has been implemented as deep-embedded domain-specific language in contrast to the shallow-embedded version of ForSyDe discussed in Sect. 4.2, which has been developed for simulation. In the deep-embedded version of ForSyDe, the system knows about its own structure, and ForSyDe's embedded compiler can operate on the abstract syntax tree to perform different analysis and transformation activities, such as simulation of the system or translation into a target language, e.g., VHDL in the case of the hardware synthesis tool. Thus there is no need for external parsers to compile a deep-embedded ForSyDe model. So far the deep-embedded version mainly supports hardware synthesis from the synchronous MoC, but using the same technique different back ends, like SW synthesis to C, can be developed within the embedded compiler.

In order to get access to the internal structure of the system model, the deep-embedded version of ForSyDe relies on several advanced Haskell techniques, which also affects the syntax of ForSyDe system models. In particular the impact of Template Haskell [55] is clearly visible in the definition of ForSyDe functions, but enables that all details of the system model are known to the embedded compiler at compile time. All examples in this section have been modeled with the `forsyde-deep` library, which is available on https://github.com/forsyde/forsyde-deep, and have been run using version 7.10.3 of the Glasgow Haskell Compiler `ghc`.

Listing 6 shows the synthesizable model of a counter in deep-embedded ForSyDe. The counter consists of a single process `counterProc`, which is created

Listing 6 A synthesizable counter in the deep-embedded version of Haskell-ForSyDe

```
1  -- Enable Language Extension: Template Haskell
2  {-# LANGUAGE TemplateHaskell #-}
3
4  module CounterHW (Direction, counterSys) where
5
6  import ForSyDe.Deep
7  import Data.Int
8
9  type Direction = Bit
10
11 nextStateFun :: ProcFun (Int8 -> Direction -> Int8)
12 nextStateFun = $(newProcFun
13    [d| nextState state dir
14           = if dir == H then
15               if state < 9 then state + 1
16               else 0
17             else
18               if state == 0 then 9
19               else state - 1
20      |])
21
22 counterProc :: Signal Direction -> Signal Int8
23 counterProc = scanldSY "counterProc" nextStateFun 0
24
25 counterSys :: SysDef (Signal Direction -> Signal Int8)
26 counterSys = newSysDef counterProc "Counter" ["direction"]
27              ["number"]
```

as finite state machine without output decoder by means of the process constructor `scanldSY` and the function argument `nextStateFun`. The reason for the special syntax inside `nextStateFun`, i.e., `$(newProcFun [d| ... |])`, is the use of Template Haskell to get access to the internal structure of the system model. Deep-embedded ForSyDe introduces the concept of *system* as a new hierarchical level. A system has a name, input and output ports, and there is full information about its internal structure, so that functions operating on the internal structure of the system can be defined for analysis, simulation or synthesis. An example is the simulation command `simulate` that enables the simulation of a system.

```
*CounterHW> simulate counterSys [L,H,H,H,H,L,L,L,L]
[0,9,0,1,2,3,2,1,0]
```

New systems can be created by instantiation of system components and their composition into a new system as illustrated in Listing 7, where a new system is created through composition of the counter and a seven-segment decoder. Please note that in order to define vectors of fixed size, which is critical for hardware

systems, a new synthesizable data type for fixed-sized vectors `FSVec` has been defined for ForSyDe. The size of the vector is a part of the type and given by a data type constructor `D`x, where x is the size of the vector.

Listing 7 Larger systems can be composed of instantiated components using a set of equations

```
module CounterSystemHW where

import ForSyDe.Deep
import CounterHW
import SevenSegmentDecoderHW
-- omitted additional import declarations

systemProc :: Signal Direction -> Signal (FSVec D7 Bit)
systemProc dir = sevenSeg
   where
      sevenSeg   = (instantiate "sevenSegDec" sevenSegDecSys)
                   counterOut
      counterOut = (instantiate "counter" counterSys) dir

system :: SysDef (Signal Direction -> Signal (FSVec D7 Bit))
system = newSysDef systemProc "system" ["in"] ["out"]
```

Listing 8 The ForSyDe synthesis tool interacts directly with the Altera Quartus tool

```
compileQuartus_CounterSystem :: IO ()
compileQuartus_CounterSystem = writeVHDLOps vhdlOps system
  where
    vhdlOps = defaultVHDLOps{execQuartus=Just quartusOps}
    quartusOps
      = QuartusOps{action=FullCompilation,
                   fMax=Just 50, -- in MHz
                   fpgaFamiliyDevice=Just ("CycloneII",
                                           Just "EP2C35F672C6"),
                   pinAssigs=[("in", "PIN_N25"),      -- SW0
                              ("resetn", "PIN_N26"), -- SW1
                              ("clock","PIN_G26"),   -- KEY[0]
                              ("out[6]","PIN_AF10"), -- HEX0[0]
                               ...
                              ("out[0]","PIN_V13") ] -- HEX0[6]
                  }
```

The deep-embedded ForSyDe tool provides a direct link to the Altera Quartus synthesis tool. The ForSyDe compiler generates the VHDL code and passes it together with optional design constraints and pin assignments to Quartus, which

generates the netlist for the circuit. Listing 8 shows the ForSyDe code for the synthesis of the counter to an Altera DE2/35 University board.

4.5 SystemC-ForSyDe

The formal definition of ForSyDe is based on the functional programming paradigm, so a pure functional language like Haskell is a perfect fit for ForSyDe. Nevertheless, the ForSyDe modeling framework is language independent. To make ForSyDe attractive for industrial designers, a SystemC version of the ForSyDe modeling framework has been created, which follows the spirit and semantics of the formal ForSyDe framework. This section gives a short overview about the nature of SystemC-ForSyDe by using the synchronous counter example from Fig. 4.4. A more detailed discussion on SystemC-ForSyDe is given in [2].

Listing 9 shows the code for the counter in SystemC-ForSyDe, which has the same structure as the corresponding Haskell model of Listing 1. The functions for the next state and the output need to be declared as side-effect-free functions, and are arguments for the combinational process constructors `scomb2` and `scomb` to create the processes `nextstate` and `outputdecode`. Here, `scomb2` and `scomb` are special versions of `comb2` and `comb`, requiring present events as inputs. Process constructors are implemented as C++ template classes where the template parameters determine the input/output types, and the constructor arguments are either values or functions. The process `del1` is created with the `delay`-process constructor. Finally, process networks can be expressed in SystemC-ForSyDe as *composite processes*, which can be regarded as netlists created by binding signals to processes through the newly introduced concept of *ports*.

A SystemC-ForSyDe model is also aware of its internal structure, which means that introspection can be used to operate on the system structure. Hence, it is possible to extract graph representations, which can be used for subsequent phases in the design flow, such as design space exploration or synthesis. Figure 4.19 shows an automatically generated graphical representation of the SystemC model from Listing 9 to illustrate the capabilities of introspection in SystemC-ForSyDe.

SystemC-ForSyDe supports a similar set of MoCs as Haskell-ForSyDe. All the MoCs discussed in the Sect. 4.2.3 are also supported by SystemC-ForSyDe libraries, which are publicly available from the ForSyDe web page [19].

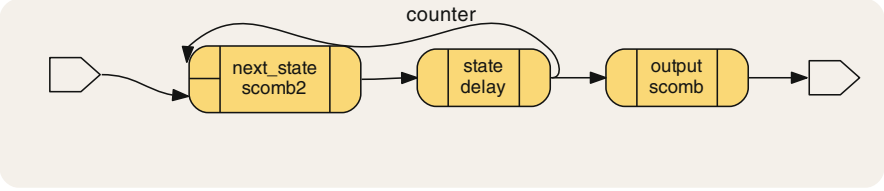


Fig. 4.19 Generated graphical representation of the counter in SystemC using the tool `f2dot`

Listing 9 The counter example in the SystemC version of ForSyDe

```
#include <forsyde.hpp>

using namespace ForSyDe;

typedef enum Direction { up, hold } Direction;
typedef enum Clock     { tick }     Clock;

void nextstate_f(int& ns, const int& s, const Direction& in) {
    switch (in) { case up   : ns = (s + 1)
                  case hold : ns = s; break; }
}

void output_f(abst_ext<Clock>& out, const int& s) {
    switch (s) { case 0  : out = abst_ext<Clock>(tick); break;
                 default : out = abst_ext<Clock>(); }
}

SC_MODULE(counter) {
   // Declaration of inputs, outputs and intermediate signals
   SY::in_port<Direction>        input;
   SY::out_port<abst_ext<Clock>> output;
   SY::signal<int>               next_state, state1, state2;

   // Module architecture: constructing processes and binding
   signals SC_CTOR(counter) {
      // Process that computes the next state
      auto nextstate = new SY::scomb2<int, int, Direction>
                                    ("next_state", nextstate_f);
      nextstate->iport1(state1);
      nextstate->iport2(input);
      nextstate->oport1(next_state);

      // Sequential process that stores the state value
      auto del1 = new SY::delay<int>("state", 0);
      del1->iport1(next_state);
      del1->oport1(state1);
      del1->oport1(state2);

      // Process that decodes the output
      auto outputdecode = new SY::scomb<abst_ext<Clock>, int>
                                        ("output", output_f);
      outputdecode->iport1(state2);
      outputdecode->oport1(output);
   }
};
```

4.6 Related Work

Researchers have for many years promoted methods that push the design entry to a higher level of abstraction by the usage of formal models and transformations in the design process. In 1997, Edwards et al. [16] expressed their beliefs "that the design approach should be based on the use of one or more formal methods to describe the behavior of the system at a high level of abstraction, before a decision on its decomposition into hardware and software is taken." Furthermore, they stated that "the final implementation of the system should be made by using automatic synthesis from this high level of abstraction to ensure implementations, that are 'correct by construction."' In 1998, Skillicorn and Talia discussed models of computation for parallel architectures in [58]. They argued that "a model must hide most of the details from programmers if they are to be able to manage, intellectually, the creation of software" and that "as much as possible of the exact structure of the executing program should be inserted by the translation mechanism (compiler and run-time system) rather than by the programmer." Furthermore, they pointed out that "models ought to be as abstract and simple as possible." Also Keutzer et al. [28] stressed that "to be effective a design methodology that addresses complex systems must start at high levels of abstraction." They promoted "the use of formal models and transformations in system design so that verification and synthesis can be applied to advantage in the design methodology" and argued that "the most important point for functional specification is the underlying mathematical model of computation."

Although the arguments for a more formal and disciplined design approach have been known for a long time, there still exists no formal and systematic design methodology that can be employed in an industrial setting. The IEEE standard language SystemC [26] has been inspired by SpecC [15] and is implemented as C++ class library. SystemC provides a discrete-event simulation kernel, and has been the base for several approaches towards a more formal design process. SystemC-AMS [59], an extension of SystemC based on a timed data-flow MoC, enables the modelling of analog and hybrid systems. HetSC [24] is based on the standard SystemC kernel and targets heterogeneous systems. HetSC supports the designer by a set of modeling primitives with additional supporting rules and guidelines for different MoCs. In comparison to ForSyDe, this approach is less formal, since it is difficult to enforce well-defined models. SysteMoC [18], which is described in more detail in ▸ Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign", is based on SystemC and addresses dynamic data-flow applications. SysteMoC's modeling technique enables to describe both statically analyzable actors and expressive dynamic actors. An actor is modeled as a state machine that separates the computational part of the actor from the processing of tokens, which is modeled as actor state machine. SysteMoC provides analysis methods operating on the actor state machine that can detect the analyzable portion of the system model, for which

powerful data-flow analysis methods exist. The Ptolemy project [17,44] studies the usage of well-defined MoCs for the design of heterogeneous, concurrent embedded and cyber-physical systems. Ptolemy uses an actor-oriented model, where actors communicate with each other by sending messages. Ptolemy introduces the concept of director, where the director defines the semantics of the underlying set of actors and thus specifies the MoC. Heterogeneous models can be created due to the concept of hierarchy, where each director defines the interaction of the actors on its own level. ForSyDe supports a generic mechanism, where each process is associated with a MoC and where the execution of the process network is only based on data dependences. Thus no central synchronization is required, which is a prerequisite for efficient simulation on parallel and distributed architectures.

Several researchers have used declarative languages to address system design from a more formal perspective. Reekie used Haskell for the modeling purpose of digital signal processing applications, where streams are modeled as infinite lists and processes are created by higher-order functions [46]. Reekie also proposed semantic-preserving transformations based on equational reasoning to convert a model into a more efficient form. The relational language Ruby has been developed for the design of hardware circuits using a structural representation of basic hardware components and connection patterns. This structured concept has been extended for software to formulate a vision on HSCD [33] based on Ruby. Lava [10] borrows many concepts from Ruby, but is embedded in the functional language Haskell. It provides a variety of powerful connection patterns, access to formal methods and translation to VHDL. There exist several versions of Lava: Chalmers Lava [10], Xilinx Lava [56], and most recently Kansas Lava [22]. The goal of Kansas Lava is to scale up the ideas in Lava to operate on larger circuits and to use larger basic components. In contrast to the structural approach of Lava and Ruby, Mycroft and Sharpe have taken a behavioral approach as base for the development of the languages SAFL (statically allocated functional language) and SAFL+ [54]. Although these languages have been primarily designed for hardware design, they have been used in [40] for HSCD. SAFL offers the application of semantic-preserving transformations and can synthesize programs in a resource-aware style, where functions that are called several times result in shared resources. The Hawk [35] language, based on Haskell, addresses the modeling, simulation and verification of microprocessors, where it exploits the formal base of Haskell. Hardware ML (HML) [31] is based on the functional programming language Standard ML [39]. It has been designed as an alternative to VHDL with a direct mapping of the HML constructs to the corresponding VHDL code. Cλash [3] is a functional hardware description language that uses a subset of Haskell for the purpose of describing hardware. Haskell functions denote components and the Cλash-compiler converts Cλash-descriptions into the corresponding hardware implementation. There are several approaches to generate target code for GPUs from Haskell-based domain-specific languages. Examples are Nikola [34] and Obsidian [60].

ForSyDe has been inspired by the work of Reekie and is based on the same modeling approach for signals and processes. The work of Mycroft and Sharp

shares the same ambition as ForSyDe to move system design to a higher level of abstraction, but followed a different modeling approach. Furthermore, refinement is restricted to semantic-preserving transformations. Cλash, Lava, Ruby, and HML are developed as languages for hardware design and operate on a lower abstraction level than ForSyDe. Their main objective is to provide a formally sound alternative to VHDL and Verilog. In contrast, ForSyDe addresses the modeling and design of embedded and cyber-physical design, and views VHDL merely as a target language. Hawk focuses on processor design with a focus on modeling and verification of instruction set and architecture, and does not support hardware synthesis. Furthermore, ForSyDe is the only declarative approach that is based on model of computation theory and that provides several models of computation and MoC interfaces.

The ForSyDe concept of process constructors is heavily influenced by the work of Skillicorn on *homomorphic skeletons* [57]. The term skeleton, coined by Cole [13] in his seminal work on *algorithmic skeletons*, has been used in the parallel programming community to denote an abstract building block that has a predefined implementation on a parallel machine. In order to obtain an implementation, the abstract program must be composed of these skeletons. The advantage of such an approach is that it raises the level of abstraction, because programmers program in their language and do not even have to be aware of the underlying parallel architecture. Specialists can be used to design the implementation of these skeletons. Using the Bird-Meertens formalism, Bird demonstrates how to derive programs from specifications by equational reasoning using lists [8], arrays, and trees [9] as data types. As Skillicorn points out, implementations with guaranteed performance can be built for computers that are based on standard topologies. Also cost measures can be provided since the complete schedule of computation and communication is known from the implementation of the skeleton.

The influential CIP (computer-aided, intuition-guided programming) project investigated transformational program construction [4]. CIP follows a top-down approach that starts with a formulation of the formal specification which is then converted via well-defined semantic-preserving transformations into a final program. The authors stated the following advantages of this approach in [4]: (a) the final program is correct by construction; (b) the transitions can be described by schematic rules and thus be reused for a whole class of problems; (c) due to formality the whole process can be supported by the computer; (d) the overall structure is no longer fixed throughout the development process, so that the approach is quite flexible.

The development of a successful practical design transformation system is a huge challenge. The transformation framework does not only need to provide a sufficient number of transformation rules, but has also to derive a sequence of transformations steps that yield a correct and efficient implementation. So far transformational approaches have mainly been used for small general purpose programming modules with a high demand on formal correctness. The problem is aggravated in the embedded systems domain, because of extra-functional

design constraints and restricted resources. Most approaches for transformational hardware design are restricted to semantic-preserving transformations [43, 53], while ForSyDe's support of nonsemantic transformations enables to integrate the refinement techniques in the area of high-level synthesis [21,37] as design decisions. The result of transformational design refinement from a high-level general purpose language is largely dependent on the initial specification due to the very large design space that can only partly explored [62]. The problem is of fundamental character and also known as syntactic variance problem in high-level synthesis [20]. The problem is naturally smaller in domain-specific languages like ForSyDe with a smaller set of building blocks, but it cannot be eliminated still exists. A more detailed overview on program transformation is given in [41], while [42] concentrates on transformation techniques for functional and logical programs.

4.7 Conclusion

The ForSyDe design methodology aims at pushing system design to a higher level of abstraction and provides means to enable a correct-by-construction design flow. ForSyDe is based on a solid formal foundation in form of a well-defined functional system model and a theoretical base in form of model of computation theory. The chapter gave an overview about the key concepts in ForSyDe and illustrated its heterogeneous system modeling using its Haskell version, which can be viewed as a perfect match with the underlying formal framework. Furthermore, the chapter also discussed how to convert an abstract system model into a final implementation by transformational design refinement followed by system synthesis. The ForSyDe framework is language-independent and can even be realized in other languages, which has been demonstrated by the SystemC version of ForSyDe that has been developed for industrial designers.

ForSyDe is an active research project that covers the whole design flow. In addition to system modeling, design refinement and system synthesis, current research focuses also on the development of a design flow for mixed-criticality multi-processor applications sharing the same platform. Here, the underlying formal models of computation and the concept of process constructors enable to create analysis models from the ForSyDe system models. These analysis models can then be used in the critical design space exploration activity to find efficient implementations on shared multi-processor platforms. A first implementation of the design space exploration (DSE) tool, which uses constraint programming and takes communication on shared resources into account, is presented in [47]. Once an efficient mapping with the corresponding schedules has been calculated by the DSE tool, the schedule and the function arguments of the process constructors need to be synthesized to the processors on the target platform. Future research will develop the synthesis concepts for multi-processor platforms based on the synthesis concepts presented in Sect. 4.4.

Appendix: Introduction to Haskell

This section gives a short introduction to functional languages and Haskell to give readers who are not familiar with functional programming additional background information. For more information on Haskell, visit the Haskell home page [61].

A functional program is a function that receives the program's input as argument and delivers the program's output as result. The main function of a program is defined in terms of other functions, which can be composed of still other functions until at the bottom of the functional hierarchy the functions are language primitives. Haskell is a pure functional language, where each function is free from side-effects. This means given the same inputs, which in case of ForSyDe could be a set of input signals, a Haskell function will always produce identical outputs. Thus the whole functional program is free from side-effects and thus behaves totally deterministic. Since all functions are free from side-effects, the order of evaluation is only given by data dependencies. But this means also that there may exist several possible orders for the execution of a functional program.

Considering the function

$$f(x, y) = u(h(x), g(y))$$

the data dependencies imply that the functions $h(x)$ and $g(y)$ have to be evaluated before $u((h(x), g(y))$ can be evaluated. However, since there is no data dependency between the functions h and g, there are the following possible orders of execution:

- $h(x)$ is evaluated before $g(y)$;
- $g(y)$ is evaluated before $h(x)$;
- $h(x)$ and $g(y)$ are evaluated in parallel.

Thus functional programs contain implicit parallelism, which is very useful when dealing with embedded system applications, since they typically have a considerable amount of built-in parallelism. Of course it is also possible to parallelize imperative languages like C++, but it is much more difficult to extract parallelism from programs in such languages, since the flow of control is also expressed by the order of statements.

In addition to common data types, such as `Bool`, `Int`, and `Double`, Haskell also defines lists and tuples. An example for a list is `[1,2,3,4] :: [Integer]`, which is a list of integers. The notation "`::`" means "has type." An example for a tuple, which is a structure of different types is `('A', 3) :: (Char, Integer)` where the first element is a character and the second one is an integer. Haskell has adopted the Hindley-Milner type system [38], which is not only strongly typed but also uses type inference to determine the type of every expression instead of relying on explicit-type declarations.

Haskell is based on the lambda-calculus and allows to write functions in *curried* form, where the arguments are written by juxtaposition. The following Haskell function `add` is written in curried form.

```
add :: Num a => a -> a -> a
add x y = x + y
```

Since '`->`' associates from right to left, the type of `add` can also be read as

```
add :: Num a => a -> (a -> a)
```

This means that given the first argument, which is of a numeric type `a`, it returns a function from `a` to `a`. This enables *partial application* of a curried function. New functions can then be defined by applying the first argument, e.g.,

```
inc x = add 1
dec x = dec 1
```

These functions only have one argument and the following type

```
inc :: Num a => a -> a
dec :: Num a => a -> a
```

Another powerful concept in functional languages is the *higher-order function*, which is adopted in ForSyDe for process constructors. A higher-order function is a function that takes functions as argument and/or produces a function as output. An example of a higher-order function is `map`, which takes a function and a list as argument and applies ("maps") the function `f` on each value in the list. The function `map` is defined as follows

```
map f []     = []              -- Pattern 1 (empty list)
map f (x:xs) = f x : map f xs -- Pattern 2 (all other lists)
```

The higher-order function `map` uses an additional feature of Haskell, which is called *pattern matching* and is illustrated by the evaluation of `map (+1) [1,2,3]`.

```
      map (+1) [1,2,3]
⇒     map (+1) (1:[2,3])          Pattern 2 matches
⇒     1+1 : map (+1) [2,3]        Evaluation of Pattern 2
⇒     2 : map (+1) (2:[3])        Pattern 2 matches
⇒     2 : 2+1 : map (+1) [3]      Evaluation of Pattern 2
⇒     2 : 3 : map (+1) (3:[])     Pattern 2 matches
⇒     2 : 3 : 3+1 : map (+1) []   Evaluation of Pattern 2
⇒     2 : 3 : 4 : map (+1) []     Pattern 1 matches
⇒     2 : 3 : 4 : []              Evaluation of Pattern 2
⇒     [2,3,4]
```

During an evaluation the patterns are tested from the top to the bottom. If a pattern, the left-hand side, matches, the corresponding right-hand side is evaluated. The expression `map (+1) [1,2,3]` does not match the first pattern since the list is not empty (`[]`). The second pattern matches, since (`x:xs`) matches a list that is constructed of a single value and a list. Since the second pattern matches, the right-hand side of this pattern is evaluated. This procedure is repeated recursively until

the first pattern matches, where the right-hand side does not include a new function call. As this example shows, lists are constructed and processed from head to tail.

The higher-order function `map` can now be used with all functions and lists that fulfill the type declaration for `map`, which Haskell infers as

```
map :: (a -> b) -> [a] -> [b]
```

Another important higher-order function is *function composition*, which is expressed by the composition operator '.'.

```
(.)       :: (b -> c) -> (a -> b) -> (a -> c)
f . g     =  \x -> f (g x)
```

This definition uses "lambda abstractions" and is read as follows. The higher-order function `f . g` produces a function that takes a value x as argument and produces the value $f(g(x))$. The expression `f = (+3) . (*4)` creates a function `f` that performs $f(x) = 4x + 3$. Function composition is extremely useful in ForSyDe since it allows to merge processes in a structured way.

Haskell allows to define own data types using a `data` declaration. It allows for recursive and polymorphic declarations. A data type for a list could be recursively defined as

```
data AList a = Empty
             | Cons a (AList a)
```

The declaration has two *data constructors*. The data constructor `Empty` constructs the empty list and `Cons` constructs a list by adding a value of type `a` to a list. Thus `Cons 1 (Cons 2 (Cons 3 Empty))` constructs a list of numbers. The term *type constructor* denotes a constructor that yields a type. In this case `AList` is a type constructor. As mentioned before, the list data type is predefined in Haskell. Here `[]` corresponds to `Empty` and `:` to `Cons`. `[a]` corresponds to `AList a`. The ForSyDe `Signal` is defined in the same way as the data type `AList`, see Sect. 4.2.1.

References

1. Acosta A (2007) Hardware synthesis in ForSyDe. Master's thesis, School for Information and Communication Technology, Royal Institute of Technology (KTH), Stockholm. KTH/ICT/ECS-2007-81
2. Attarzadeh Niaki S, Jakobsen M, Sulonen T, Sander I (2012) Formal heterogeneous system modeling with SystemC. In: Forum on specification and design languages (FDL 2012), Vienna, pp 160–167
3. Baaij C, Kooijman M, Kuper J, Boeijink A, Gerards M (2010) Cλash: structural descriptions of synchronous hardware using Haskell. In: 2010 13th Euromicro conference on digital system design: architectures, methods and tools (DSD), pp 714–721
4. Bauer FL, Möller B, Partsch H, Pepper P (1989) Formal program construction by transformations – computer-aided, intuition guided programming. IEEE Trans Softw Eng 15(2):165–180
5. Benveniste A, Berry G (1991) The synchronous approach to reactive and real-time systems. Proc IEEE 79(9):1270–1282
6. Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, Simone RD (2003) The synchronous languages 12 years later. Proc IEEE 91(1):64–83

7. Berry G, Gonthier G: The Esterel synchronous programming language: design, semantics, implementation. Sci Comput Program 19(2):87–152 (1992)
8. Bird RS (1986) An introduction to the theory of lists. Technical monograph PRG-56 edn. Oxford University Computing Laboratory
9. Bird RS (1988) Lectures on constructive functional programming. Technical Monograph PRG-69 edn. Oxford University Computing Laboratory
10. Bjesse P, Claessen K, Sheeran M, Singh S (1998) Lava: hardware design in Haskell. In: International conference on functional programming, Baltimore, pp 174–184
11. Blindell GH, Menne C, Sander I (2014) Synthesizing code for GPGPUs from abstract formal models. In: Forum on specification and design languages (FDL 2014), Munich
12. Boussinot F, De Simone R (1991) The Esterel language. Proc IEEE 79(9):1293–1304
13. Cole M (1989) Algorithmic skeletons: structured management of parallel computation. Research monographs in parallel and distributed computing. Pitman, London
14. Derler P, Lee E, Sangiovanni-Vincentelli A (2012) Modeling cyber-physical systems. Proc IEEE 100(1):13–28
15. Dömer R, Gerstlauer A, Gajski D (2002) SpecC language reference manual, version 2.0
16. Edwards S, Lavagno L, Lee EA, Sangiovanni-Vincentelli A (1997) Design of embedded systems: formal models, validation, and synthesis. Proc IEEE 85(3):366–390
17. Eker J, Janneck J, Lee E, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y: Taming heterogeneity – the Ptolemy approach. Proc IEEE 91(1):127–144 (2003)
18. Falk J, Haubelt C, Teich J (2006) Efficient representation and simulation of model-based designs in SystemC. In: Proceedings of the forum on specification and design languages (FDL), vol 6, pp 129–134
19. ForSyDe: Formal system design. https://forsyde.ict.kth.se/
20. Gajski DD, Ramachandran L (1994) Introduction to high-level synthesis. IEEE Des Test Comput 11(4):44–54
21. Gajski DD, Dutt ND, Wu ACH, Lin SYL (1992) High-level synthesis. Kluwer Academic, Boston
22. Gill A, Bull T, Kimmell G, Perrins E, Komp E, Werling B (2010) Introducing kansas Lava. In: Morazán M, Scholz SB (eds) Implementation and application of functional languages. Lecture notes in computer science, vol 6041. Springer, Berlin/Heidelberg, pp 18–35
23. Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous data flow programming language Lustre. Proc IEEE 79(9):1305–1320
24. Herrera F, Villar E (2008) A framework for heterogeneous specification and design of electronic embedded systems in SystemC. ACM Trans Des Autom Electron Syst 12(3): 22:1–22:31
25. Herrholz A, Oppenheimer F, Hartmann PA, Schallenberg A, Nebel W, Grimm C, Damm M, Haase J, Brame J, Herrera F, Villar E, Sander I, Jantsch A, Fouilliart AM, Martinez M (2007) The ANDRES project: analysis and design of run-time reconfigurable, heterogeneous systems. In: International conference on field programmable logic and applications (FPL'07), pp 396–401
26. IEEE Standard for Standard SystemC Language Reference Manual. IEEE Std 1666–2011 (Revision of IEEE Std 1666–2005), pp. 1–638. http://ieeexplore.ieee.org/document/6134619/
27. Jantsch A (2005) Models of embedded computation. In: Zurawski R (ed) Embedded systems handbook. CRC Press, Boca Raton. Invited contribution
28. Keutzer K, Malik S, Newton AR, Rabaey JM, Sangiovanni-Vincentelli A (2000) System-level design: orthogonolization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543
29. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
30. Lee EA, Sangiovanni-Vincentelli A (1998) A framework for comparing models of computation. IEEE Trans Comput-Aided Des Integr Circuits Syst 17(12):1217–1229
31. Li Y, Leeser M (2000) HML, a novel hardware description language and its translation to VHDL. IEEE Trans VLSI 8(1):1–8

32. Lu Z, Sander I, Jantsch A (2002) A case study of hardware and software synthesis in ForSyDe. In: Proceedings of the 15th international symposium on system synthesis, Kyoto, pp 86–91
33. Luk W, Wu T (1994) Towards a declarative framework for hardware-software codesign. In: Proceedings of the third international workshop on hardware/software codesign, Grenoble, pp 181–188
34. Mainland G, Morrisett G (2010) Nikola: embedding compiled GPU functions in Haskell. In: Proceedings of the third ACM Haskell symposium on Haskell, Haskell '10. ACM, New York
35. Matthews J, Cook B, Launchbury J (1998) Microprocessor specification in HAWK. In: International conference on computer languages (ICCL'98), pp 90–101
36. McKinley PK, Sadjadi SM, Kasten EP, Cheng BH (2004) Composing adaptive software. IEEE Comput 37(7):56–64
37. Micheli GD (1994) Synthesis and optimization of digital circuits. McGraw-Hill, New York
38. Milner R (1978) A theory of type polymorphism in programming. J Comput Syst Sci 17: 348–375
39. Milner R, Tofte M, Harper R, MacQueen D (1997) The definition of standard ML – revised. MIT, Cambridge
40. Mycroft A, Sharp R (2000) Hardware/software co-design using functional languages. In: Proceedings of tools and algorithms for the construction and analysis of systems (TACAS). LNCS, vol 2031. Springer, pp 236–251
41. Partsch HA (1990) Specification and transformation of programs. Springer, New York
42. Pettorossi A, Proietti M (1996) Rules and strategies for transforming functional and logic programs. ACM Comput Surv 28(2):361–414
43. Plosila J (1999) Self-timed circuit design – the action systems approach. PhD thesis, University of Turku, Turku
44. Ptolemaeus C (ed) (2014) System design, modeling, and simulation using Ptolemy II. Ptolemy.org. http://ptolemy.org/books/Systems
45. Raudvere T, Sander I, Jantsch A (2008) Application and verification of local non-semantic-preserving transformations in system design. IEEE Trans Comput-Aided Des Integr Circuits Syst 27(6):1091–1103
46. Reekie H (1995) Realtime signal processing. PhD thesis, School of Electrical Engineering, University of Technology at Sydney
47. Rosvall K, Sander I (2014) A constraint-based design space exploration framework for real-time applications on MPSoCs. In: Design automation and test in Europe (DATE '14), Dresden
48. Sander I (2003) System modeling and design refinement in ForSyDe. PhD thesis, Royal Institute of Technology, Stockholm
49. Sander I, Jantsch A (2004) System modeling and transformational design refinement in ForSyDe. IEEE Trans Comput-Aided Des Integr Circuits Syst 23(1):17–32
50. Sander I, Jantsch A (2008) Modelling adaptive systems in ForSyDe. Electron Not Theor Comput Sci (ENTCS) 200(2):39–54
51. Sangiovanni-Vincentelli A (2007) Quo vadis, SLD? Reasoning about the trends and challenges of system level design. Proc IEEE 95(3):467–506
52. Schneider K (2009) The synchronous programming language Quartz. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern
53. Seceleanu T (2001) Systematic design of synchronous digital circuits. PhD thesis, University of Turku, Turku
54. Sharp R, Mycroft A (2001) A higher level language for hardware synthesis. In: Proceedings of 11th advanced research working conference on correct hardware design and verification methods (CHARME). LNCS, vol 2144. Springer, pp 228–243
55. Sheard T, Jones SP (2002) Template meta-programming for Haskell. ACM SIGPLAN Not 37(12):60–75
56. Singh S, James-Roxby P (2001) Lava and JBits: from HDL to bitstream in seconds. In: Proceedings of the 9th annual IEEE symposium on field-programmable custom computing machines, FCCM '01, pp 91–100

57. Skillicorn DB (1994) Foundations of parallel programming. Cambridge international series on parallel computation. Cambridge University Press, Cambridge/New York
58. Skillicorn DB, Talia D (1998) Models and languages for parallel computation. ACM Comput Surv 30(2):123–169
59. Standard SystemC AMS extensions 2.0 language reference manual (2013)
60. Svensson J, Sheeran M, Claessen K (2011) Obsidian: a domain specific embedded language for parallel programming of graphics processors. In: Scholz SB, Chitil O (eds) Implementation and application of functional languages. Lecture notes in computer science, vol 5836. Springer, Berlin/Heidelberg, pp 156–173
61. The Haskell language. http://www.haskell.org
62. Voeten J (2001) On the fundamental limitataions of transformational design. ACM Trans Des Autom Electron Syst 6(4):533–552

Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design Approach

5

Fernando Herrera, Julio Medina, and Eugenio Villar

Abstract

Model-based design has shown to be a powerful approach for embedded software systems. The Unified Modeling Language (UML) provides a standard, graphically based formalism for capturing system models. The standard Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile provides syntactical and semantical extensions required for the modeling and HW/SW codesign of real-time and embedded systems. However, the UML/MARTE standard is not sufficient. In addition, a modeling methodology stating how to build a model capable to support the analysis and HW/SW codesign activities of complex embedded systems is required. This chapter presents a UML/MARTE modeling methodology capable to address such analysis and design activities. A distinguishing aspect of the modeling methodology is that it supports a *single-source* design approach.

Acronyms

BCET	Best-Case Execution Time
BSP	Board Support Package
CPS	Cyber-Physical System
DSE	Design Space Exploration
DSL	Domain-Specific Language
EDF	Earliest Deadline First
EML	Execution Modeling Level
ESL	Electronic System Level

F. Herrera • E. Villar
GESE Group, TEISA Department, ETSIIT, Universidad de Cantabria, Santander, Cantabria, Spain
e-mail: fherrera@teisa.unican.es; evillar@teisa.unican.es

J. Medina (✉)
Software Engineering and Real-Time Group, University of Cantabria, Santander, Cantabria, Spain
e-mail: medinajl@unican.es

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_6

FPGA	Field-Programmable Gate Array
GME	Generic Modeling Environment
HLS	High-Level Synthesis
HRM	Hardware Resource Modeling
HSCD	Hardware/Software Codesign
ISA	Instruction-Set Architecture
M2M	Model-to-Model
MARTE	Modeling and Analysis of Real-Time Embedded Systems
MCS	Mixed-Criticality System
MDA	Model-Driven Architecture
MPSoC	Multi-Processor System-on-Chip
NFP	Non-Functional Property
OMG	Object Management Group
OS	Operating System
PIM	Platform Independent Model
PVT	Programmers View Time
RR	Round Robin
RTOS	Real-Time Operating System
SLS	System-Level Synthesis
TLM	Transaction-Level Model
UML	Unified Modeling Language
UTP	Universal Testing Profile
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VSL	Value Specification Language
WCET	Worst-Case Execution Time

Contents

5.1 Introduction

Model-based design is a powerful approach for the design of complex embedded systems [27]. It can be adapted to different design contexts and domains, being compatible with methodologies like *Agile* [2]. The Unified Modeling Language (UML) supports Model-Driven Architecture (MDA) [40] and provides the following remarkable advantages:

- It is a **widely spread** language, **known and used in different domains**.
- It is an Object Management Group (OMG) **standard** [30].
- It provides a set of **generic modeling elements**, supported by a **graphical syntax** and a closed set of **diagrams**, which enables the capture of architectural and behavioral details.

A key of the success of UML is related to the generality of the provided modeling elements. These elements have a simple semantics that can be easily understood and interpreted by engineers of different domains. For instance, stating that a UML *port* is a mechanism to access to/from a UML *component* which hides component internals. This simple element semantics is encompassed by a simple graphical syntax, which facilitates the comprehension and adoption of UML diagrams.

UML has been also proposed for the modeling of embedded systems. Embedded systems have become complex. The increasing amount of silicon available in a single-chip enables the integration of more hardware and software functions. In close relationship, the specification and modeling tasks have become increasingly complex. Models need to reflect systems integrating multiple applications and diverse software platform components, e.g., embedded RTOS, middleware, drivers, etc. Similarly, current hardware architectures rely on multi-core processors, surrounded by many HW devices for communication, storage, sensing, and actuation. In addition, several types of analysis are applied (e.g., schedulability, timed-simulations, etc.) which require to add additional information to the model, e.g., annotations of extra-functional properties related to timing, memory sizes, energy, etc.

A solution is to build models integrating parts under different Domain-Specific Languages (DSLs). However, fragmentation into DSLs limits the understanding of the overall model (all the engineers handling the model should know all the DSLs involved) and requires an additional effort to integrate the DSLs.

In this scenario, relying on UML has the advantage of providing a common and comprehensive host language. UML lacks the semantics and specific elements required for tackling the Hardware/Software Codesign (HSCD) of complex

embedded systems. However, UML enables to cover this *semantic lack* by means of a UML extension mechanism called profile. A UML profile provides *stereotypes*. Stereotypes are applied to UML modeling elements and add them additional attributes and domain-specific semantics. In fact, the OMG currently provides a rich portfolio of UML profiles oriented to different domains like telecommunication, middleware, and real time.

Among these OMG standard profiles, the Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile [29] provides a rich set of modeling elements, sufficiently broad to cover HSCD of real-time and embedded systems. For instance, it supports the modeling of the hardware and software platform, of the application, of extra-functional properties, and the definition of performance and real-time analyses.

As well as the UML and the MARTE profile, a modeling methodology is required. The modeling methodology states how to use the language in order to build the model of a system. It states which information have to be captured and how it is structured and captured through the available modeling techniques, such that the model can be processed and provides all the information required by the analysis tools and processes related to the HW/SW codesign. All these aspects defining a modeling methodology have to serve to its main purposes. The main purpose of the methodology presented in this chapter is to enable the development of abstract models which can be used as a single-source for the different activities involved in electronic system-level design. The modeling methodology shall not fix a specific Electronic System Level (ESL) design flow, but it has to allow that applicable design flows can progressively enrich the model from early stages, when limited information is available, toward an enriched model which allows the implementation of an efficient, potentially optimal, solution.

Following, Sect. 5.2 presents the goals of the methodology. The section motivates how these goals turn into requirements because of the need of higher design productivity. Moreover, it precises the meaning of some of this requirements (e.g., single-source, incremental modeling) in the modeling context. Section 5.3 provides an overview of related modeling approaches, showing how they partially cover the aforementioned requirements. Then, Sect. 5.4 presents the generic UML-based modeling techniques adopted by the methodology to cover Sect. 5.2 requirements. Section 5.5 introduces, before the conclusions, a single-source design framework exploiting the presented modeling methodology.

5.2 Modeling Requirements

5.2.1 Single-Source Approach

A model-driven design approach helps in analyzing and predicting the behavior of a system from different perspectives and for different purposes. In a *multi-source* approach, several models of the same system have to be developed, each one associated to a perspective, type of analysis, or design activity. However, relying on several

models easily leads to extra modeling efforts, redundancies, inconsistencies, and traceability problems. In contrast, an advanced software development methodology like *Agile* adopts a *single-source* approach, which reduces the maintenance burden and the traceability burden and increases consistency [2].

The same advantages motivate the proposal of a single-source approach for modeling and design of embedded systems [7], which is sketched in Fig. 5.1.

A single model, a UML/MARTE model in this case, captures all the information required by the many tasks involved in a modern embedded system design flow, e.g., reusability, verification, schedulability analysis, architectural mapping, simulation and performance analysis, design space exploration, etc.

The left-hand side of Fig. 5.2 illustrates the multi-source approach, where two independent models, A and B, of an embedded system are developed.

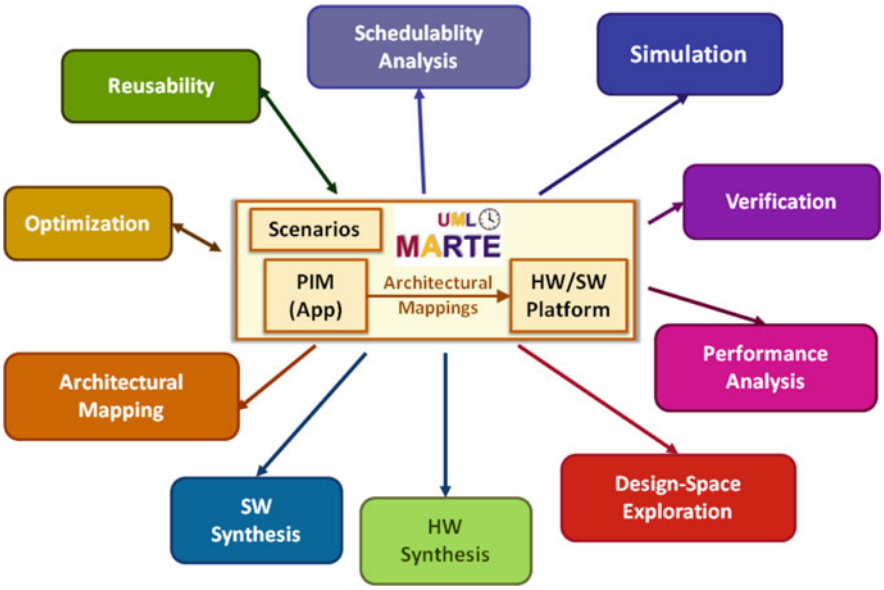


Fig. 5.1 The MARTE model as a single-source model

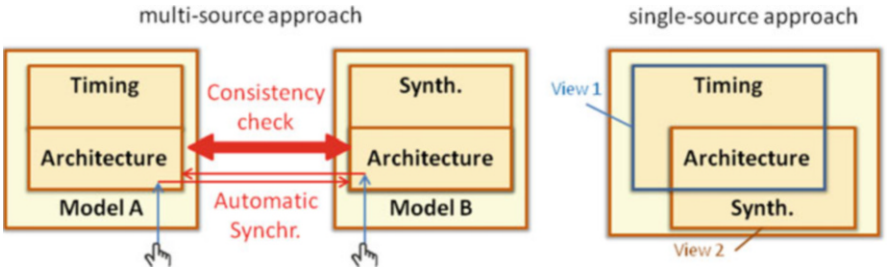


Fig. 5.2 Multi-source vs. single-source approach

Model A serves for analyzing timing performance. Model B is the input for a SW synthesis process. Model A adds some timing annotations, which are not present in model B. Model B adds target information, which is not present in model A. However, both models should reflect the same architecture of the application. Therefore, a double unnecessary modeling effort has been done. A consistency check to ensure that both models A and B reflect the same architecture is required, at least after the first development of the models and specially in a context where model architectures can be edited. These problems exponentially grow with the number of independent models required (in the worst case, one per design activity shown in Fig. 5.1).

In contrast, in the single-source approach (right-hand side of Fig. 5.2), the architectural information is captured once in the model. Then, in order to capture models A and B, the model is extended twice: once for capturing the time annotations required by the A model and once more for capturing the synthesis information required by the B model. The adoption of a single-source approach is a distinctive and remarkable aspect of the shown approach. Moreover, there are other important characteristics which need to be preserved and adopted, as addressed in the following sub-sections.

5.2.2 Separation of Concerns

The single-source approach centralizes the information required for the design tasks in a single model, with the advantages discussed in the previous section. The complex system model will have a big amount of information. In this context, *Separation-of-concerns* helps to provide a structure to such information. Model *Viewpoints* [22] and *Perspectives* [46] become essential for the modeling activity. Viewpoints are about defining the most relevant categories of information and about the structure the model concerning them. Perspectives are related to the model concerns, e.g., model properties or actions performed on the model, that shall be presented and can be accessed in the modeling framework. Combining perspectives and viewpoints facilitates model edition. It enables the cooperation of different modelers, which can focus on specific information of the model. It also simplifies tools and activities around it, e.g., the model navigation performed by code generators.

5.2.3 Incremental Modeling

In software engineering, *incremental modeling* refers to the delivery of a series of releases called *increments* [39]. This approach enables to progressively provide more functionality on each increment. In software modeling, increments can refer to customizations, as well as to extensions [1].

In our single-source modeling context, incremental modeling refers to the possibility to start by building a first model that enables a first set of analysis

and design activities, and enhance it later on. Then, further modeling increments may enable additional design activities and/or improve the results of the previously applicable activities, e.g., enabling more accuracy in the performance assessments. A model increment shall not prevent performing the design activities already enabled by previous versions of the model.

5.2.4 Component-Based Functional Modeling

A software component-based approach [38] has important advantages. First, it enables to build system functionality as a composition of existing and reusable components. These components interact with each other only through well-defined interfaces [32], which declare the functional services they provide and require. Adopting a software-centric approach, i.e., assuming a default software implementation of the functionality, is also an efficient approach according to the increasing and dominant amount of software in current embedded systems.

5.2.5 Support of System-Level Design Activities

Adoption of ESL design [4] is key in order to shorten the *productivity gap* [19]. ESL design tackles the productivity gap by raising the abstraction of the starting model and by introducing automated design activities around that system-level model. Two main design activities are:

- **Design Space Exploration (DSE):** consists in the activity of exploring design alternatives and selecting the optimal ones.
- **System-Level Synthesis (SLS):** consists in the generation of the implementation from the initial model, used as a specification. It requires the decision of the HW/SW partition and the automated generation of software, hardware, and HW/SW interfaces.

DSE can rely on SLS. For instance, [4] highlights that one of the most important benefits for High-Level Synthesis (HLS) is that it enables (hardware) design exploration. The methodology in [37] enables the automated SW synthesis from a UML/MARTE model of the binaries targeted to a multi-core heterogeneous platform. This automation is exploited for the exploration of different software implementation alternatives. The aforementioned approaches are applied after HW/SW partition.

An *explore-then-synthesise* approach relying on performance assessment techniques is also possible. Kang et al. [20] refers to DSE as the activity of exploring design alternatives *prior to implementation*. Therefore, in this approach, the exploration activity is done first on the model. Performance estimations on different model configurations, which can reflect different HW/SW partitions, serve to decide the most convenient one. Thus the system-level synthesis is only applied on that

choice. In a UML/MARTE context, [13] showed how the model can reflect different mappings, comprising different HW/SW partitions. It enables the generation of a performance assessment model which, linked to a DSE tool, enabled an automated search of the Pareto solutions set.

The automation of the DSE and SLS activities is key in order to cope with the exploration of huge design spaces and in order to eliminate human errors both in the exploration and implementation steps. This has motivated the development of automated DSE, SW synthesis, and hardware HLS frameworks. The possibility to exploit these frameworks relies on the fact of enabling an input model which conjugates the abstraction required by ESL, with the information required for performing these activities. This chapter shows how it is done from a UML/MARTE model under a single-source approach.

5.2.6 Support of Mixed-Criticality

Mixed-Criticality System (MCS) have got an increasing interest [23]. MCS integrate applications with constraints whose fulfillment has different levels of criticality and which can share computational, memory, and communication resources. Although there is not an unanimous convey in the meaning of criticality, it has been associated to the impact of the occurrence of a failure [6]: e.g., *safety critical* when the failure can cause injures to humans, *mission critical* when the failure prevents the system to perform its expected behavior, but it does not compromise safety, and *low critical* when the impact is affordable. Safety standards associate criticalities to a set of more or less strict requirements on the development process. There are several reasons for the highest interest in mixed-criticality systems. Reusing existing functionalities and integrating them in the same chip or in the same platform is an efficient way to exploit growing integration capabilities and cost-effective platforms. However, dependability problems arise, as not all the functionalities and performance requirements are equally important. A mixed-criticality aware design methodology considers this differences. For instance, methodologies need to combine real-time analysis for the safety parts with hard real-time constraints, with other techniques employed in the optimization of soft real-time embedded systems, e.g., based on simulation and on average-case optimization. Accordingly, MCS models need to provide support for the emerging mixed-criticality design methodologies. Mixed-criticality has to be reflected in system models.

The modeling scenarios which can be identified in recent MC research and in current industrial practices related to safety standards lead to the need to associate criticalities to different type of elements, i.e., application components, platform resources, constrains, annotations of extra-functional properties. These criticality annotations are used to apply mixed-criticality-specific modeling rules to feed mixed-criticality-specific schedulability analyses, DSE flows, and development and implementation constraints.

5.3 State of the Art

There are several examples of model-based methodologies for modeling, exploration, and implementation of complex embedded systems. For instance, *MILAN* [5] enables a model-based approach to capture the application and the platform (called resource model). MILAN also introduced the idea of constraint model, which distinguishes between semantic constraints, which give composability rules, from design constraints, which capture performance requirements. The framework shows how a model-based approach facilitates the integration of several simulation tools at different levels of abstraction for the estimation of the performance of the design point by relying on a Generic Modeling Environment (GME) [35]. *Koski* [21] is a design flow for Multi-Processor System-on-Chip (MPSoC) covering the design phases from system-level modeling to Field-Programmable Gate Array (FPGA) prototyping. System-level modeling relies on UML and separates the capture of the application from the platform. However, this approach relies on a proprietary profile for capturing the required semantics.

Despite the relative recent release of the standard MARTE profile, there have been already several proposals relying on it. *Gaspard2* [8, 35] is a design environment for data-intensive applications which enables a MARTE description of both the application and the hardware platform, including MPSoC, and regular structures. Gaspard2 uses composite diagrams and the MARTE profile for capturing both application and platform architectures. Gaspard2 tooling supports the chaining of different Model-to-Model (M2M) transformation tools. This facilitates the generation of synthesis flows and also of performance models. Specifically, Gaspard2 supports the generation of SystemC TLM models at the Programmers View Time (PVT) level. It enables fast simulations, which speeds up exploration.

MoPCoM [45] is another design methodology for the design of real-time embedded systems which supports UML and the MARTE profile for system modeling. Specifically, MoPCoM uses the Non-Functional Property (NFP) MARTE profile for the description of real-time properties; the Hardware Resource Modeling (HRM) MARTE profile for platform description; and the Alloc MARTE profile for architectural mapping. Moreover, MoPCoM defines three levels of generation. The second level, called Execution Modeling Level (EML), targets the generation of models for performance analysis, and it is suitable for obtaining performance figures used in DSE iterations. However, work reported in [24] mostly focuses on the Detailed Modeling Level (DML), intended for implementation, by enabling VHDL code generation.

The *PHARAON* methodology [33] provided a solution for automatically synthesizing models combining new communication semantics with standard UML/MARTE real-time management features. This approach provides a flexible and easy-to-use way to specify and explore the system's concurrent architecture.

CoFluent methodology [18] captures application and hardware architecture by means of composite diagrams and SysML blocks. UML activity diagrams are used to specify application execution flows. The MARTE HRM profile is used for capturing the HW platform. CoFluent models can be translated into executable SystemC transaction-level models, which serves to obtain utilization, time, and power performance metrics.

A main limitation of the previous methodologies is that the exploration of architectural alternatives requires the edition of the UML/MARTE model and a re-generation of the executable performance model.

In [24], a UML-MARTE-based methodology relying on activity threads is proposed in order to reduce the effort required to capture the set of architectural mappings. An activity thread is a UML activity diagram where each path reflects a design alternative, that is, an architectural mapping.

In [26], a methodology for supporting designers on the evaluation of the HW/SW partitioning solutions, specifically, to identify design points fulfilling the timing constraints is shown. It proposes a way to depict in one set of diagrams all possible combinations of system configurations. By means of annotation of MARTE non-functional properties and of the application of schedulability analysis, the design space is restricted to the design points fulfilling timing requirements. However, this methodology does not rely on automated technologies for the estimation of performance metrics.

These MARTE-based specification methodologies are still limited for DSE purposes. The exploration of different platform architectures, of different architectural mappings, and even a small change in a design parameter (e.g., a cache size) still requires a manual change of the model. The *COMPLEX* [13] flow proposes a single-source approach to overcome the aforementioned limitations. Moreover, the COMPLEX framework produces a configurable performance model which avoids the re-generation and re-compilation of a performance model for each exploration alternative and thus a significant impact in the exploration time. This framework also supported the capture of the output performance metrics to be used by the objective function(s) of the DSE process within the model and of performance constraints. Enabling the capture of the performance metrics in the model, and so in a tool independent manner, enabled the direct relation of such metrics with the performance constraints also captured in the model.

Modeling complexity has increased with the need to consider the modeling of Cyber-Physical System (CPS) and Mixed-Criticality Systems (MCS). A *UML-Modelica-SysML* integrated modeling environment as a ModelicaML profile integrated in Eclipse is presented in [36]. *Modelica* is an object-oriented mathematical language for component-oriented modeling of complex physical systems containing components of diverse nature, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power, etc. The modeling of mixed-criticality systems in UML/MARTE has been proposed in [15, 16]. This work provides modeling techniques which cover a number of scenarios where mixed-criticality has to be captured.

5.4 Single-Source Modeling Methodology

5.4.1 Introductory Example: Quadcopter System

Along the following sections, the main modeling techniques of the proposed single-source methodology are presented. These modeling techniques will be presented by taking excerpts of a model of a digital electronics system embedded in a quadcopter. This quadcopter system, shown in Fig. 5.3, has been developed by the OFFIS Institute for Information Technology in the context of the CONTREX project [28]. The quadcopter digital system includes:

- The data mining, radio control & telemetry, and flight control functionalities. They are safety critical as a failure on them compromises person's safety.
- A mission functionality, which consists in the detection and tracking through a camera a moving ball, e.g., in a sport action, in order to track, record, and stream that action video to a base station.
- A functionality to log monitoring and debug data.

All this functionality is implemented in a *Xilinx Zynq* platform, which contains a processing system with two *ARM Cortex-A9* processors. In addition, the Zynq platform has an FPGA which allows the integration of custom logic and additional *Microblaze* processors, configured without caches, for enabling more predictable computational resources for safety critical functions. The Zynq board is integrated

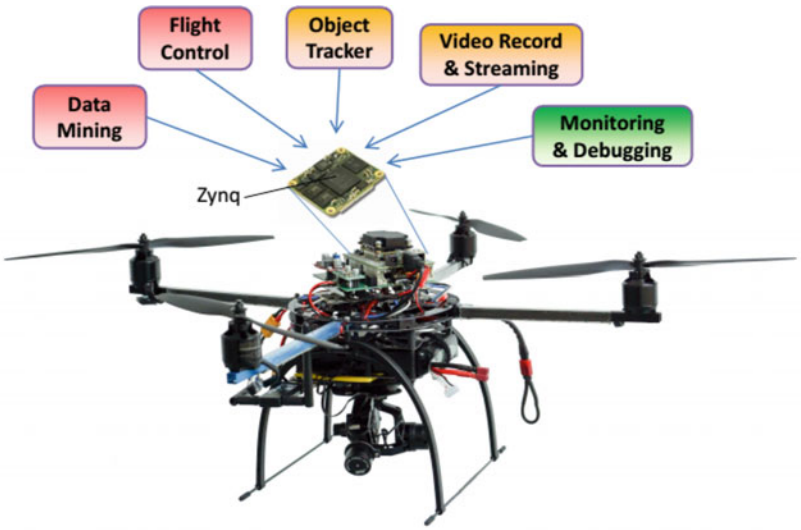


Fig. 5.3 The digital electronic system of a quadcopter is used for introducing the single-source methodology

in a board together with sensor components, motor actuators, and IO devices IO. These components are abstracted as part of the environment in the UML/MARTE model.

5.4.2 Introduction

The modeling methodology supports separation of concerns. At the root of the UML/MARTE model, model information is distributed into *views*. Model views are captured as UML packages decorated with a methodology-specific stereotype which adds the view semantics. Figure 5.4 shows a diagram with the views enclosing all the quadcopter model information.

In this methodology, model views support the separation under different concerns. The model of the verification environment (*verification view*) is separated from the system model (remaining views). As stated in MDA, the Platform Independent Model (PIM) is separated from the platform model. The PIM is captured through the data, functional and application views. Platform-dependent information is added later. The *HW resources view* declares the HW components, which can be later instanced in the *architectural view*. The declaration of software platform resources (in the *SW platform view*) is also separated from the declaration of hardware platform resources. Architectural information is captured through UML composite diagrams, and separated from component declaration, which have associated the behavioral information. The methodology follows a pragmatical and generic approach with respect to the association of behavior to the model. Figure 5.5 shows an excerpt of the quadcopter model where source files are associated to application components (UML artifacts allocated to the PIM component via UML relations decorated with the MARTE «allocated» stereotype). In addition, the methodology allows the association of paths for the sources through UML

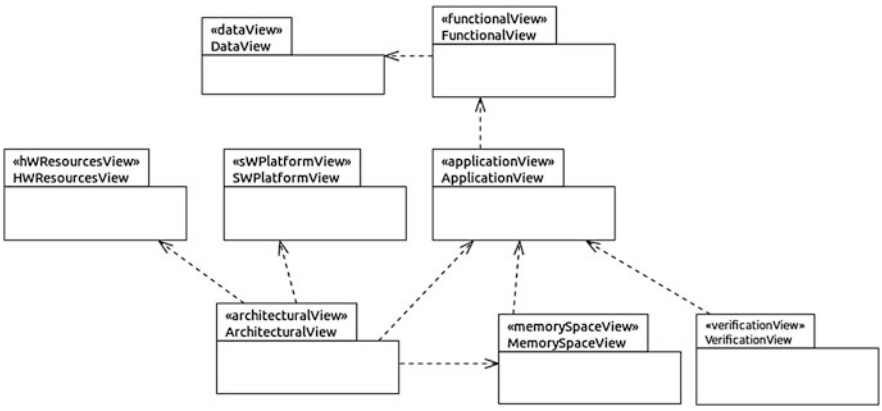


Fig. 5.4 Model views

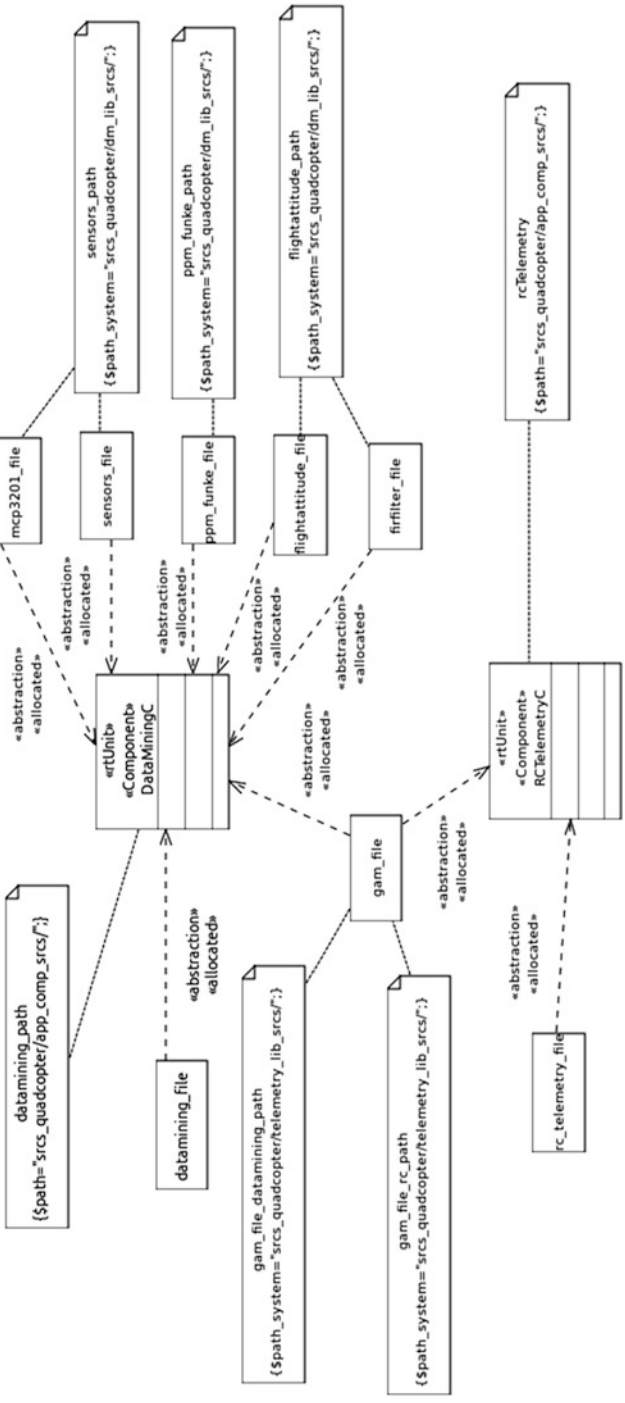


Fig. 5.5 Association of source files containing the behavior to PIM components

constraints associated to the artifacts. This mechanism is language independent and has been exploited in the automated generation of executable performance models and of synthesized binaries.

As Fig. 5.4 shows, building a platform-specific model depends on the PIM model, but not the opposite. Then, incremental model development is possible. Thus, for instance, the methodology makes possible the generation of an executable functional model. After capturing the platform-dependent model, the generation of a performance model or the synthesis phase is enabled. Similarly, time or energy annotations can be added, e.g., to a hardware component, to add accuracy to the automatically generated performance model. However, if these annotations are not present, the generation of the performance model is still possible (default values are used).

5.4.3 Platform-Independent Model

The methodology enables the description of a platform-independent model (PIM) under a component-based approach.

The methodology enables a clean separation of the PIM information. Figure 5.6 shows the components of the quadcopter PIM as they are seen in the Eclipse model explorer view.

Among the six components, the *quadcopter_app* component is decorated as *«system»* component (it can be seen in Fig. 5.7). The PIM system component is the top component of the PIM model hierarchy, which contains the PIM architecture, shown in Fig. 5.7. The remaining components are PIM components, to be eventually instanced in the PIM system component and which shall be stereotyped to be either an active component, a passive component, or a shared variable. The MARTE *«RtUnit»*, *«PpUnit»*, and *«SharedComResource»* stereotypes are respectively used for that purpose. Figure 5.8 shows the application of the *«RtUnit»* stereotype to the datamining (DataMiningC) and radio-control and telemetry (RCTelemetryC) components of the quadcopter. As can be observed in Fig. 5.6, non-system components have been enclosed in an additional UML package called *AppComponents*. This

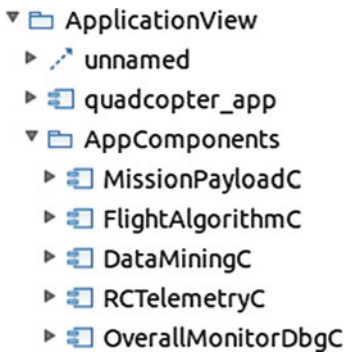


Fig. 5.6 Components of the quadcopter PIM model

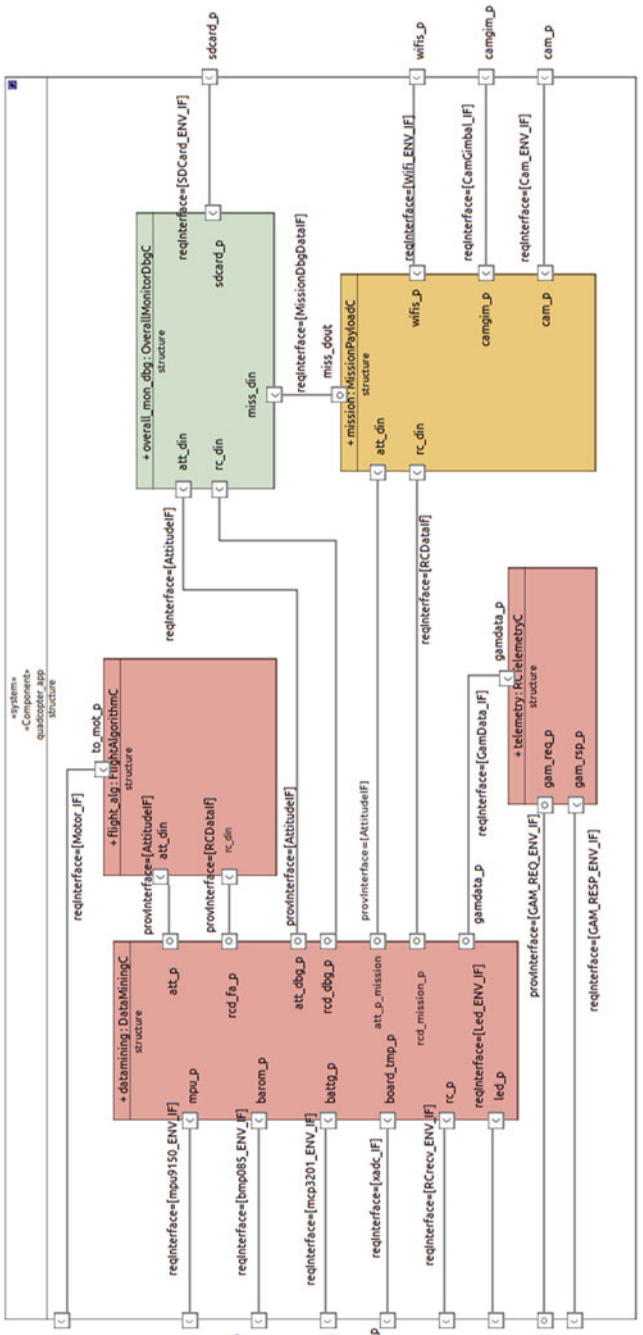


Fig. 5.7 Quadcopter PIM architecture

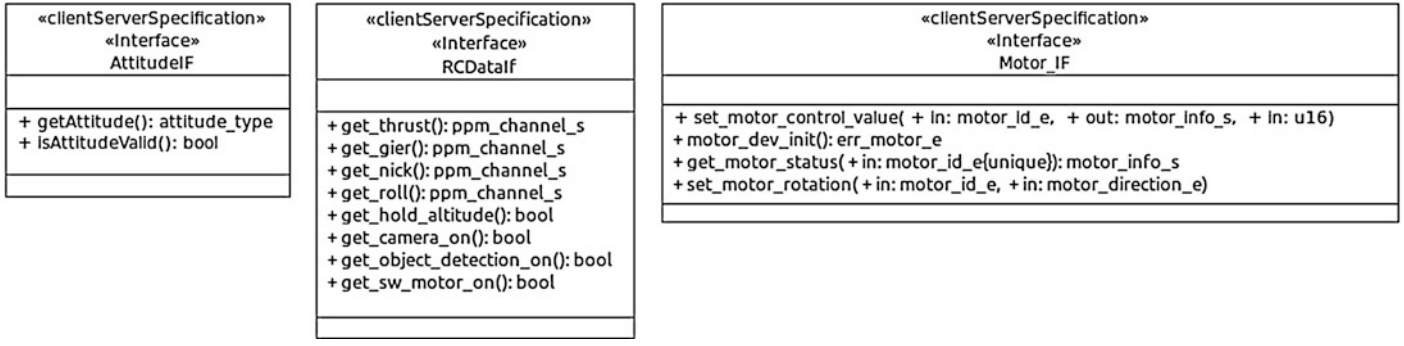


Fig. 5.8 Quadcopter functional view excerpt

is not required, but supported by the methodology, to structure more PIM model information.

The architecture of the platform-independent model is captured within the PIM system component by means of a UML composite diagram, as shown in Fig. 5.7.

The PIM system component makes instances of the PIM components as UML properties, either active components or passive components, that is either of *«RtUnit»* or *«PpUnit»* components. The quadcopter model only instances active components, since all the components have internal periodic tasks. For instance, *datamining* is an instance of the *DataMiningC* component. PIM component instances are connected via channels. Channels are captured as UML port-to-port connectors. While these connectors reflect to *internal connections*, the PIM architecture also contains UML connectors. In this case, they connect a port of the top component and a port of an internal component, reflecting the delegation of the function services or requirements across one hierarchy level.

Each port is stereotyped as a client-server port via the MARTE *«ClientServerPort»* stereotype. This stereotype has the attribute *kind*, of the MARTE *ClientServerKind* type, which allows the methodology to state that the port has either a *provided* or a *required* interface. In addition, the attributes *provInterface* and *reqInterface*, both of the UML interface type, enable the specification of the specific interface associated to the port. Such an interface has to be previously captured as a client-server interface in the functional view. Figure 5.8 shows an excerpt of the functional view of the quadcopter, with three interfaces.

All of them have been applied to the MARTE *«ClientServerSpecification»* stereotype to identify the interfaces that can be exported by PIM components. Each interface declares the methods that are exported at that interface. For instance, the AttitudeIF interface declares two methods: the *getAttitude* method for obtaining the attitude information from the provider component and the *isAttituteValid* method for obtaining a flag stating if the attitude value that can be currently retrieved from the component is valid. In turn, each of those methods are specified by their input and output parameters. Both of them have to have a precisely specified type. The data view enables the user to precise all the types to be employed in the interfaces. Figure 5.9 shows an excerpt of the data view of the quadcopter model. This excerpt shows the capability of the methodology to capture complex structured types, e.g., the *attitude_type* returned by the *getAttitude* method.

Concerning the capture of communication semantics, Fig. 5.7 reflects the simplest case, where a default semantics is associated to channels. For instance, the default semantics states that the call to the service is blocking at the initiation and end of the service. That is, the component requiring the service, i.e., making the call, waits for the provider component, i.e., the one implementing the called function, to be ready for executing it, and also waits for its completion.

In addition, the methodology enables a more detailed specification of the channel semantics. Figure 5.10 illustrates a case where the semantics of the channel used by the *overall_mon_dbg* component to retrieve attitude data from the *datamining* component has a user-defined semantics. For it, the UML port-to-port connector representing the channel is stereotyped with the *«channel»* methodology-specific

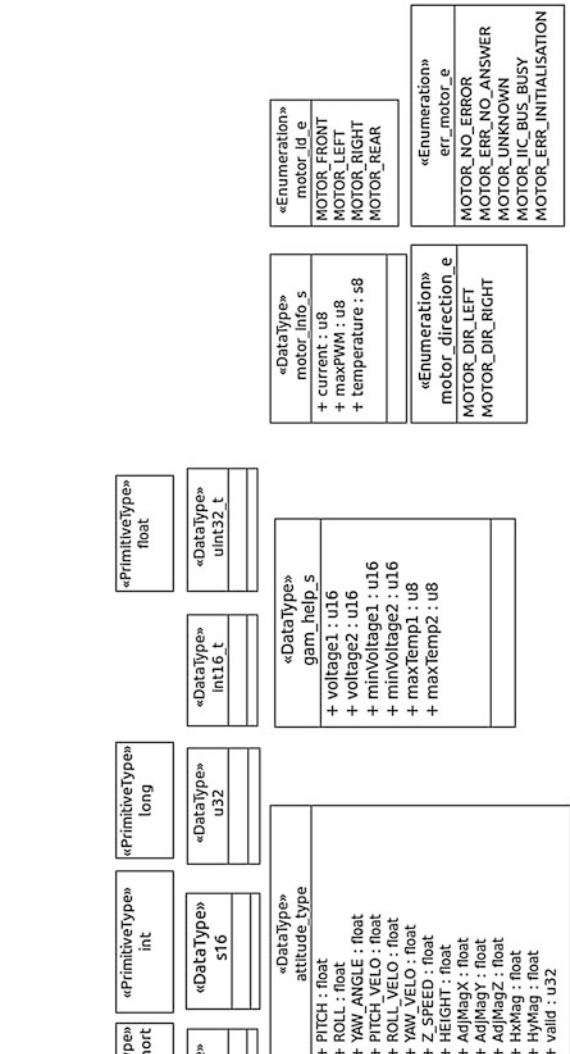


Fig. 5.9 Quadcopter data view excerpt

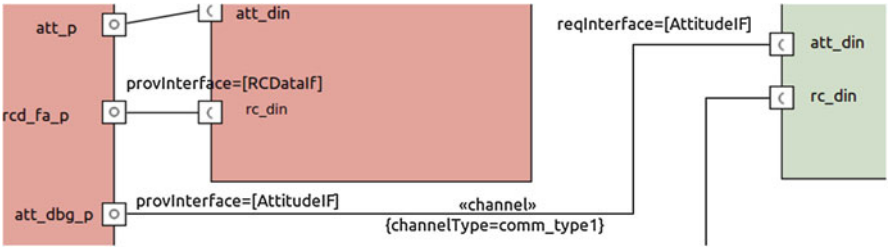


Fig. 5.10 Channel instance with custom semantics

Fig. 5.11 Specification of channel with custom semantics

stereotype. The *«channel»* stereotype provides the attribute *channelType*. This attribute can be assigned a UML component decorated with the MARTE *«CommunicationMedia»* stereotype. This component has to be included in an additional view, the communication view (see Fig. 5.11), and represents a channel whose semantics can be customized by the user. In order to customize the communication semantics, the MARTE *«StorageResource»* stereotype and the *«ChannelTypeSpecification»* stereotype are used. The former stereotype makes possible to specify the channel buffering capability, i.e., how many function calls can be buffered by the channel. The methodology-specific *«ChannelTypeSpecification»* stereotype contributes additional attributes for configuring the channel semantics, for instance, if the communicating components shall synchronize at function call and at function return.

5.4.4 Platform Resources

As was mentioned, the methodology enables the specification of platform resources in two separated views. The *SW platform view* is used to declare the software resources of the platform, i.e., operative systems and drivers. Figure 5.12 shows the software platform resources in the software platform view of the quapcopter model. The basic SW platform resource is the Operating System (OS), which is captured as a UML component decorated by the methodology-specific *«OS»* stereotype. The only OS semantics injected by this means is used for producing performance models based on generic RTOS models. However, more specific information is required in other contexts. The *«OS»* stereotype provides the *type* property, which serves as a string descriptor which uniquely identifies the target OS or RTOS in SW synthesis. The methodology also supports a more detailed specification of the OS behavior. This is necessary for safety critical cases, where an accurate performance analysis also relies on an accurate modeling of the OS scheduler behavior. For it, the *«OS»*

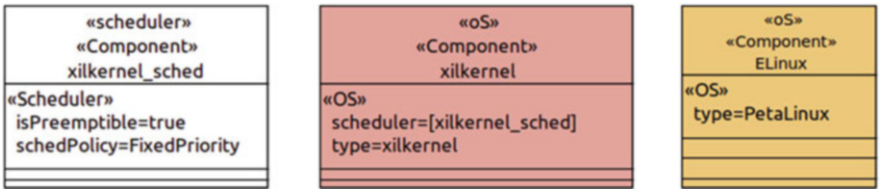


Fig. 5.12 SW resources of the quadcopter platform

stereotype provides the *scheduler* attribute. The scheduler attribute specifies one scheduler component, i.e., a component stereotyped with the MARTE *«Scheduler»* stereotype. In turn, the *«Scheduler»* stereotype enables the attributes *isPreemptible*, *schedPolicy*, *schedule*, and *otherSchedPolicy*, which enable quick and versatile specification of scheduling policy. The *schedPolicy* attribute enables a synthetic capture of the most usual scheduling policies (static scheduler, fixed priorities, Earliest Deadline First (EDF), Round Robin (RR), etc.). The attribute *isPreemptible* states that the RTOS re-schedules on release events of other application tasks. The *schedule* attribute is used to configure and complete the description of the scheduling policy when *schedPolicy*=*TimeTableDriven*. The*TimeTableDriven* value can be used in MARTE to specify both order-based schedules and time-triggered schedules. In the former case, the *schedule* attribute serves to capture the execution order, i.e., the schedule of the tasks allocated to the OS. The *otherSchedPolicy* attribute is used to support other scheduling policies not covered by MARTE. In the methodology, it is also exploited for enabling a more synthetic description capable to preserve the single-source approach in a DSE context (an example is given in [16]). In the quadcopter case, the SW resource view shown in Fig. 5.12 states that the xilkernel OS has been configured with a priority-based scheduling policy.

Figure 5.13 shows the resources declared in the hardware platform view of the quapcopter model. In this view, all the hardware platform resources to be instanced in the hardware platform architecture shall be declared. Each platform resource is declared through a component with a specific MARTE stereotype adding the hardware resource semantics. For this, the HRM MARTE profile is intensively used. As shown in Fig. 5.13, the methodology supports the modeling of computational resources (e.g., HW processors), communication resources (e.g., buses), memory resources (e.g. cache memories), main memories, and I/O devices. Depending on the type of hardware component, and thus of the stereotype, different attributes are available. None of the platform views contain any architectural information. There is one exception in the HW resources view, which allows to directly link a set of cache components to a processor component. The set is passed as the value of the *caches* attribute of the MARTE *«HwProcessor»* stereotype. Each element of the set passed to the *caches* attribute is a component decorated with the *«HwCache»* MARTE stereotype and also declared in the HW resources view. This mechanism has been used in the quadcopter model to simplify the capture of level 1 caches associated to the *ARM_Cortex_A9* processor components. For the same example,

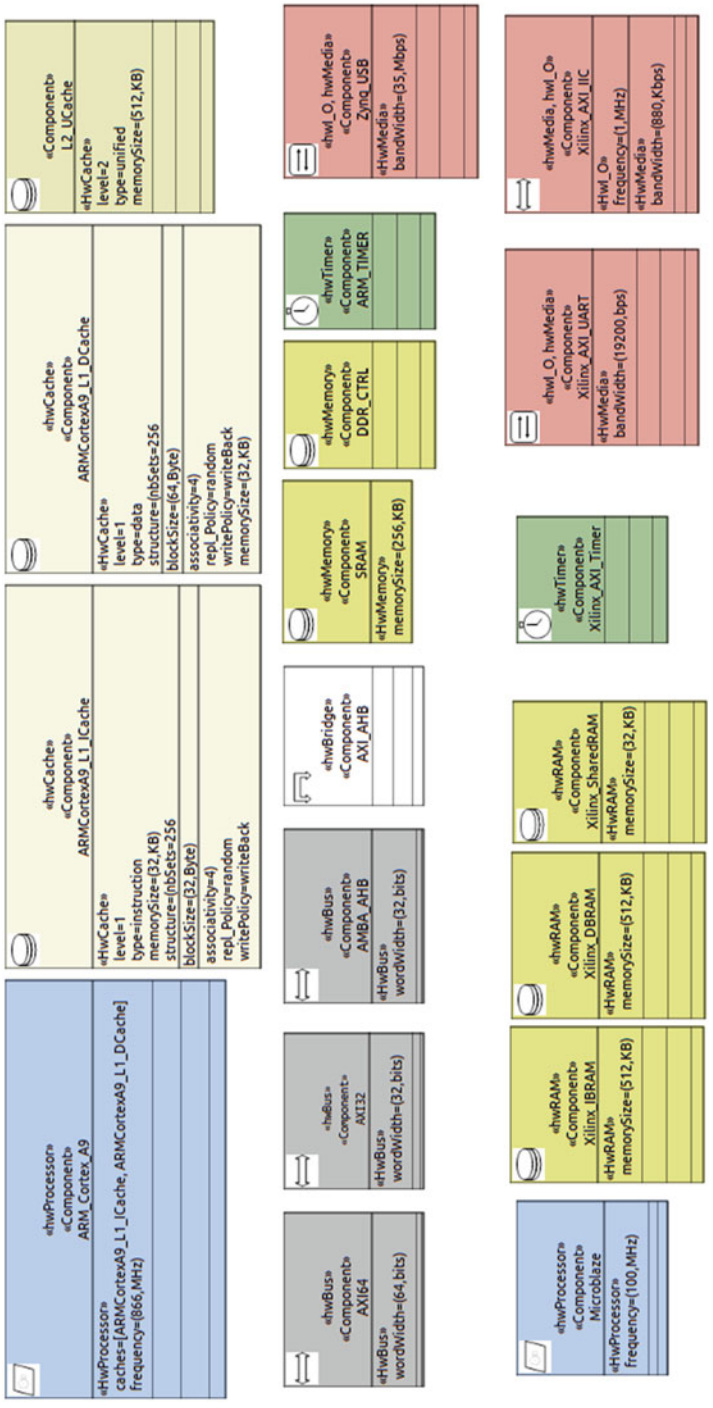


Fig. 5.13 HW resources of the quadcopter platform

the *microblaze* processor components declared in the HW resources view have an empty *cache* attribute. The microblaze processors in the quadcopter system have no cache since they are used to run the datamining and control functionalities, which require more predictability due to their criticality.

5.4.5 Platform-Specific Model

As was shown in Sect. 5.4.4, the SW platform and the HW resources views declare platform resources. They do not contain architectural information, apart from cache resources associated to processors. The methodology supports the specification of the SW/HW platform architecture and of the mapping of the PIM model to the SW/HW platform through the memory space and architectural views.

The memory space view is a non-mandatory view which can be used to specify memory spaces and the mapping of the component instances to the declared memory spaces. This is a first mapping level, which is relevant in SW implementation. A software process is inferred for each memory space. By default, if the user does not specify a memory space view, an implicit one with a single memory space is inferred, and it will be assumed that all the component instances are mapped to the implicit memory space. Figure 5.14 reflects a case where four memory spaces have been specified. The specification is done again within a composite diagram associated to a system top component, called *quadcopter_memspaces*, captured within the memory space view. This component is captured as a specialization of the PIM top component. Therefore, the references to the component instances, i.e., *telemetry*, *datamining*, *flight_alg*, *mission*, and *overall_mon_dbg*, are visible and can be used as source of the allocations.

Figure 5.15 shows the architectural view of the quadcopter system. The architectural view captures the mapping of memory spaces to OS instances, which effectively closes the mapping of the PIM to the SW/HW platform.

The SW/HW architecture captures the mapping of the OS instances onto the computing elements, i.e., HW processors, and the interconnection of the different HW elements.

As can be noticed, the mapping of memory spaces to OS instances and the mapping of OS instances to computing elements are captured again by means of UML abstractions with the *«allocate»* stereotype. Regardless of the type of source or destination, a static mapping is captured with the same modeling technique. For the connection of PIM component instances, hardware platform components are linked also through UML port-to-port connectors. Summing up, the methodology employs the same modeling techniques, to solve the same type of modeling needs, for yielding more understandable models and an easier to learn methodology.

5.4.6 Extra-Functional Properties and Performance Constraints

The methodology supports the annotation of several types of extra-functional properties (EFPs). These annotations are used by performance and schedulability

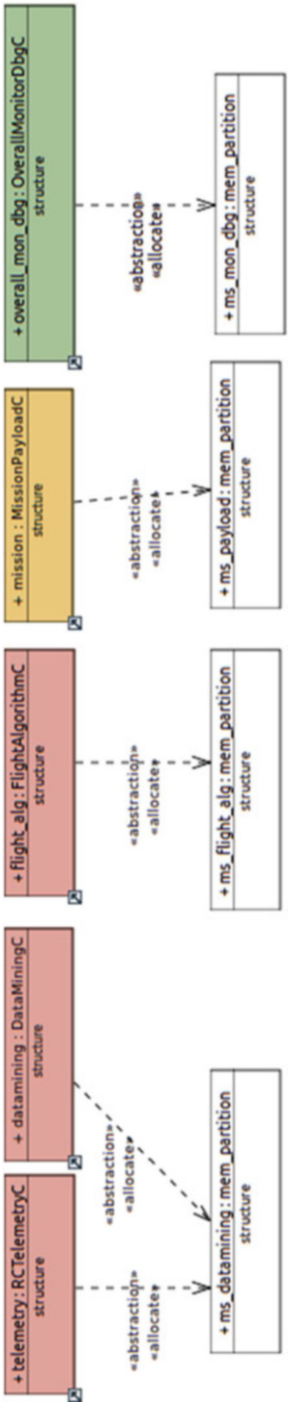


Fig. 5.14 Mapping of PIM component instances to memory partitions

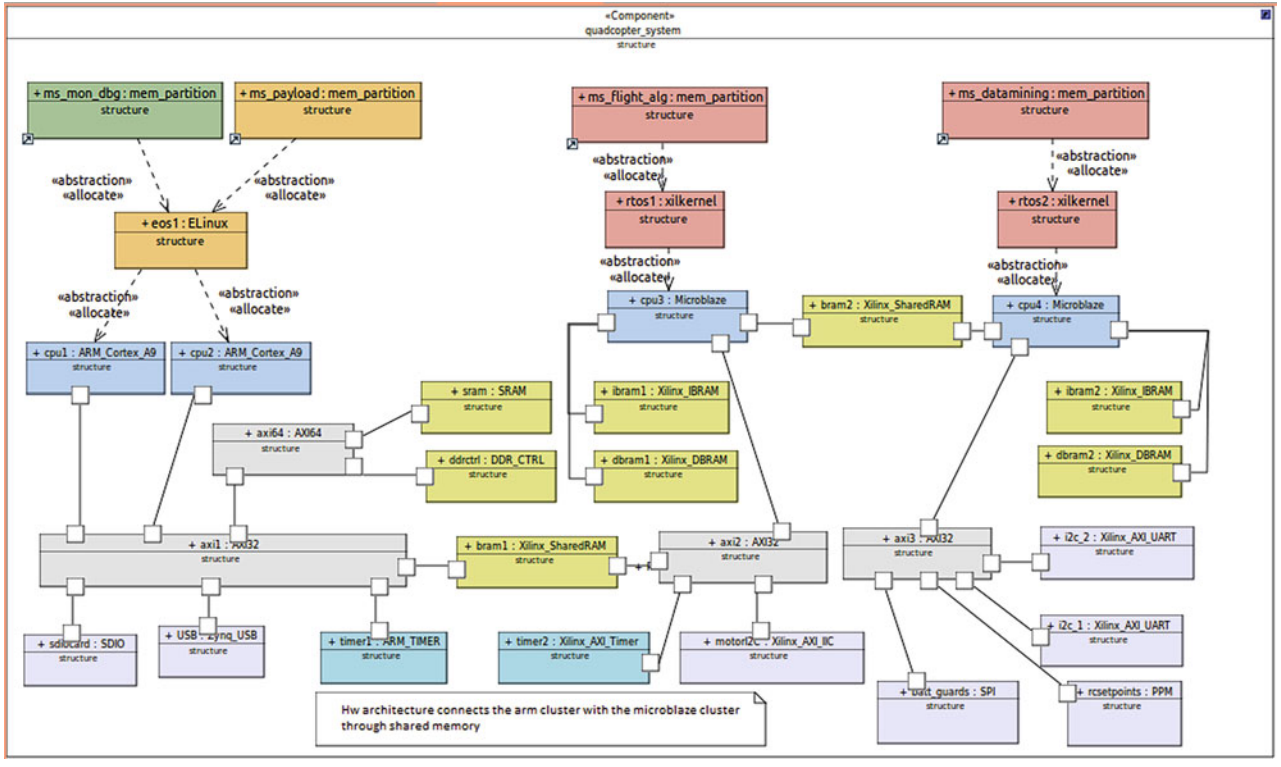


Fig. 5.15 Quadcopter architectural view

analyses. Most of them are performed in the HW resources view. Figure 5.12 illustrates several types of EFPs. For instance, the sizes of the different types of memories and structural information of caches, i.e., number of sets, cache policies, bus widths, frequencies for processors (and other types of hardware resources), and I/O device bandwidths, can be stated by only relying on the different MARTE stereotypes from the HRM profile.

Moreover, the methodology also supports the annotation of power and energy consumption associated to HW resources. For annotating static power consumption, the HW component is decorated with the MARTE *«HW_Component»* stereotype, which provides the *staticConsumption* property. Moreover, the methodology supports the modeling of power state machines. The HW component can have different functional modes defined by the operating frequency, source voltage, dynamic power, and average leakage. A UML state diagram associated to the HW component and the MARTE *«mode»* and *«ModeTransition»* stereotypes are employed for that purpose. The *«HwPowerState»* stereotype introduced in [3] and the MARTE *«ResourceUsage»* stereotypes are used for characterizing the static and dynamic power consumptions of each mode. The modeling methodology also supports the annotation of energy consumption. The annotations depend on the modeling element. For instance, in order to annotate the energy consumed per cycle in a processor, a *cycle* attribute of MARTE *NFP_Energy* type, and decorated with the MARTE *«NFP»* stereotype, is added to the processor component. An analog technique is used for associating energy consumptions to other components. For buses and memories, the energy associated to an access is annotated. For caches, two energy consumption figures are annotated, for distinguishing the *hit* consumption from the *miss* consumption.

The methodology also allows the annotation of workloads, e.g., Worst-Case Execution Time (WCET), average times, and Best-Case Execution Time (BCET), to application components. Figure 5.30 in Sect. 5.4.10 illustrates such annotations in the mixed-criticality context and how they depend on the allocation to platform resources.

The methodology enables the specification of performance requirements. Performance requirements are used in performance analysis and design space exploration to check if the assessed solutions are acceptable. Similarly, implementation and test phases should consider this information for the validation of the chosen design alternative. Figure 5.16 illustrates a compact mechanism for performance constraint specification, possible whenever the performance constraint is captured through a UML property. In the case of Fig. 5.16, through a UML comment decorated with the MARTE, *«RtSpecification»* stereotype is linked to the PIM component instances of the quadcopter. The *«RtSpecification»* contributes the *relDl* stereotype, a UML property which enables a synthetic capture of the deadline through an expression under the MARTE Value Specification Language (VSL).

The methodology provides a mode general mechanism to express performance constraints. It is done by means of a UML constraint decorated with the MARTE *«NFP_constraint»* and «ExpressionContext» stereotypes. The latter enables the capture of a Boolean expression written in VSL. The Boolean expression refers

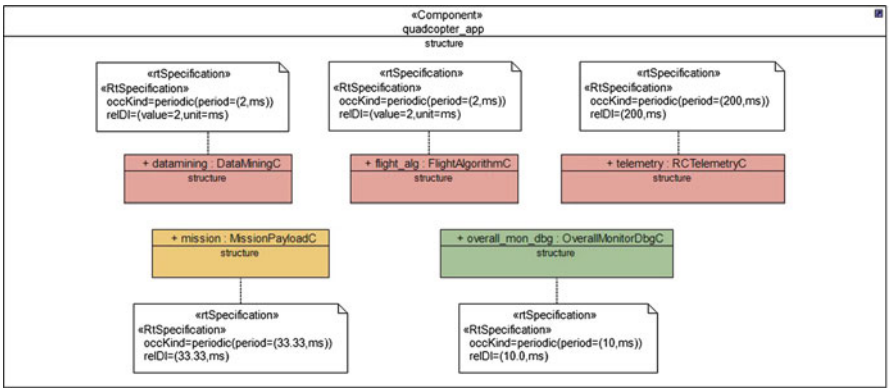


Fig. 5.16 System and application performance constraints on the quadcopter

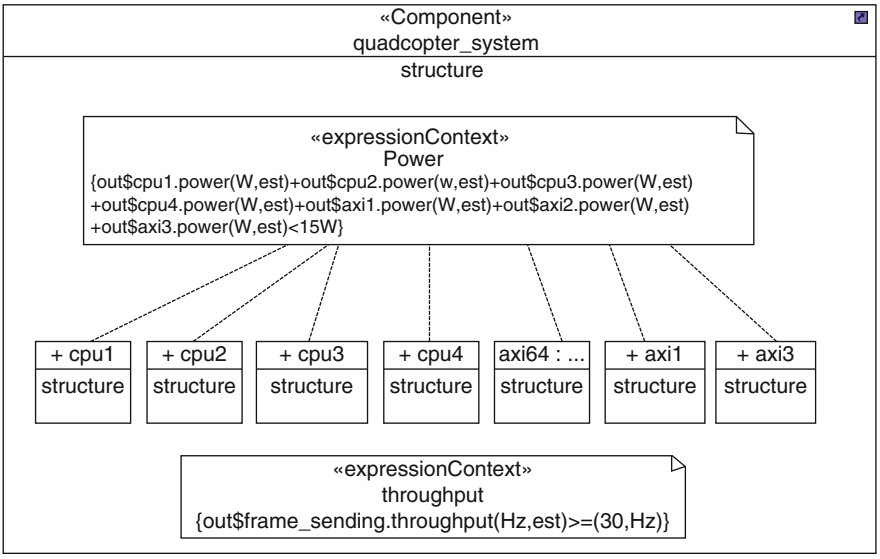


Fig. 5.17 System and application performance constraints on the quadcopter

to at least one output performance metric. An output performance metric is a value of an extra-functional property which should be estimated by an analysis tool after taking the own UML/MARTE model as an input. The output performance metric is expressed as a VSL variable with the *out* prefix. The output performance metric can be general (independent on the model), e.g., overall power consumption, or refer to a model element, e.g., a processor consumption. Figure 5.17 provides an example on the quadcopter. The NFP constraint is linked to the processor and bus instances referenced in the expression and whose power dissipation contributes to the overall power consumption. Several output performance metrics are required to

be calculated, i.e., the dissipation of each CPU and of the *axi64*, *axi1*, and *axi3* bus instances. Notice that the performance requirements refer to both application and platform elements.

5.4.7 Design Space

The proposed methodology enables the description of a design space. That is, instead of a design solution, the modeler can capture several or many design solutions, e.g., several PSMs, in a single model, thus effectively enabling the single-source approach for DSE. As was mentioned, this is crucial for avoiding model re-factoring and thus enabling fast iterations during the design space exploration.

The modeling of the design space refers to two basic modeling elements: property values annotations and architectural mappings.

The methodology supports DSE parameters. A DSE parameter is a property value annotation which, instead of specifying a single fixed value, specifies a range of possible values. The capture of a DSE parameter relies on VSL. The syntax of the VSL expression capturing the DSE parameters is the following one:

$DSEParameterName = DSERangeSpecification

The *"$"* symbol prefixes a VSL variable and thus the DSE parameter name. The *DSERangeSpecification* expresses the range of the DSE parameter, that is, all the values that the *DSEParameterName* variable can have during the exploration. The DSE parameter range can be annotated either as a collection or as an interval. Collections are captured with the syntax *DSERangeSpecification=(v1, v2, v3, unit)*, where *"v1, v2, v3"* are the numerical values of the parameter and *unit* expresses the physical unit associated the values. MARTE provides a rich set of unit kinds to support the different extra-functional properties characterizing systems components, e.g., frequencies, bandwidth, data size, etc. Intervals follow the syntax *"DSERangeSpecification=([v_{min} … v_{max}], unit)"*. For a complete determination of the exploration range, this style obliges to assume an implicit step. For an explicit and complete determination of the DSE range, the support of the style *"DSERangeSpecification=([v_{min} … v_{max}, step], unit)"* is proposed, which means a minor extension of VSL. The definition of non-linear ranges is possible. For instance, *step* can take the value *exp2*, which enables the definition of a geometrical progression, i.e., the second value is *"$v_{min}x2$"*, and so on.

Figure 5.18 illustrates the specification of a design space on the frequencies of the ARM processing cores of the quadcopter HW platform. This way, the exploration of the impact on performance depending on the selection of a Z-7020 device (which works at 667 MHz), a Z-7015 device (works at 766 MHz), or a Z-7020 device (at 866 MHz) can be explored.

In Fig. 5.18, the DSE has been associated to the processor component declaration (in the HW platform resources view). Therefore, once the DSE parameter value is fixed, it is fixed for all its instances. The methodology also allows the association of the DSE parameter to the instance properties. Therefore, if the user wants to explore the variations on the frequency of a single ARM core, then the constraint has to be

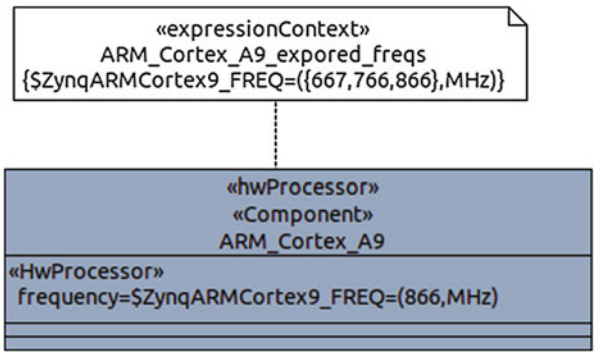


Fig. 5.18 A DSE parameter associated to the ARM Cortex-A9 processor component declaration

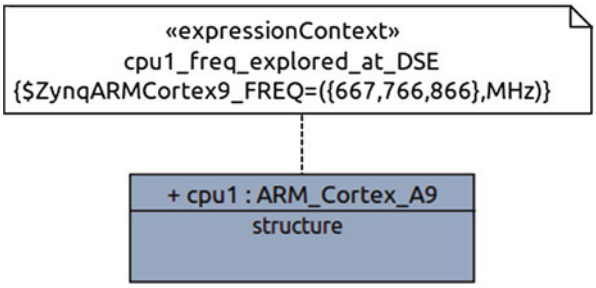


Fig. 5.19 A DSE parameter associated to the ARM Cortex-A9 processor component declaration

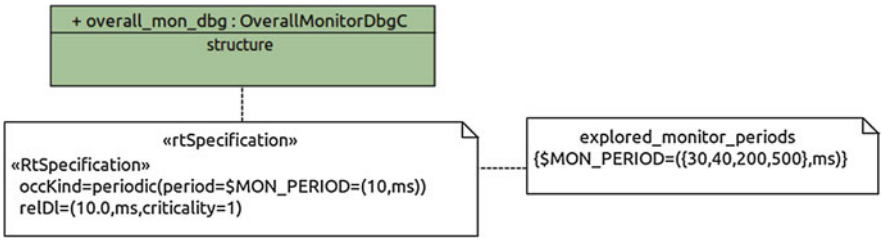


Fig. 5.20 A DSE parameter associated to the period of the monitor functionality in the quadcopter PIM

linked to that processor instance in the suitable view, i.e., in the architectural view in this case, as illustrated in Fig. 5.19.

The DSE parameter is a powerful mechanism since it is applicable to any property of the model, and therefore, it enables to specify the exploration of its impact on the performance of the system. This includes also elements of the platform-independent model. Figure 5.20 shows an example where a DSE parameter is associated to the period of the monitor and debugging component of the quadcopter PIM. In this example, this DSE parameter is useful to explore how a relaxation on its refresh frequency helps in the fulfillment of time performance constraints.

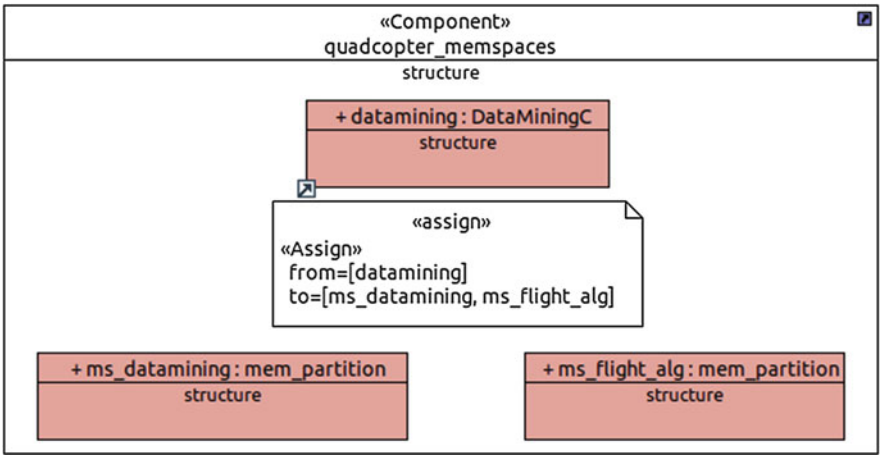


Fig. 5.21 Specification of a mapping design space in the memory space view of the quadcopter

The methodology supports the modeling of configurable mappings and, for each configurable mapping, the expression of which is the set of mappings to be explored. This way, the single-source approach is kept, as the same model serves to express all the mapping alternatives to be explored. A configurable mapping is expressed through a UML comment decorated with the MARTE *«assign»* stereotype. Specifically, its *from* and *to* attributes reflect a range of possible source and destination elements, respectively. Therefore, the *from* and *to* values are DSE parameters. Figure 5.21 shows an example of the quadcopter memory space view which specifies the exploration of two possible mappings of the *datamining* component. The *«assign»* comments can be present in the model together with the *«allocate»* fixed mappings. In a DSE context, the *«assign»* comment overrides the fixed allocation, stating the range of sources and destinations. Since the *«assign»* comment in Fig. 5.21 states nothing about the mapping of the *flight_alg*, *telemetry*, *mission*, and *overall_mon_dbg* component instances, the fixed mapping information stated in Fig. 5.14 is used in a DSE context. In an implementation context, the fixed allocations are used as a statement of the selected implementation.

Figure 5.21 illustrated the specification of a space of mappings in the memory space view (from PIM components to memory spaces). The methodology enables the modeling of configurable mappings in other levels, e.g., for the mapping of memory spaces to platform resources in the architectural view.

Several *«assign»* comments can be present in the same model. The cross product of the mapping spaces defined by each *«assign»* comment states the overall mapping space of the model.

The modeling methodology supports *DSE rules*. DSE rules are constraints embedding Boolean expressions which impose dependencies among the values of the DSE parameters. DSE rules can also refer to the *from* and *to* properties of the *«assign»* comments, and thus to configurable mappings. DSE rules provide an

additional expressiveness to the modeler to customize and prune a potentially huge and redundant design space that can result out of the cross product of all the DSE parameters and configurable mappings.

5.4.8 Modeling for Software Synthesis

The functionality associated to the model is platform independent and has no call to a specific OS API or communication middleware. In the proposed methodology, such type of platform-dependent code is automatically generated by a SW synthesis tool called eSSYN [44]. As well as the associated functionality, eSSYN needs to read PIM information, i.e., retrieved from all PIM views (data, functional and application view). In general, the PIM information read refers to the component partition, to the configuration of static and dynamic concurrency, to the semantics of the communication stated among components and to timing. Taking this information at its input, eSSYN generates all the target-dependent code to implement in SW the specified concurrency, communication, and time semantics. Most of the pieces of target-dependent code are component wrappers, named *wrappers* in short.

The partition into components determines the structure of the generated code and the ambits of visibility. Moreover, the mapping of the application component instances to memory partitions is also read at software synthesis time. A software process is inferred for each memory partition, and all functionality of PIM components mapped to the memory partition will be integrated and in the ambit of the inferred process.

Component attributes stating the concurrency semantics include the *«RtUnit»* attributes *isDynamic* and *main*. The former states if dynamic threads are created for attending the service calls. The latter captures the functionality to be statically triggered, i.e., as an autonomous thread at the beginning of the execution. The communication semantics and the mappings specified by the model have also involvements on the inference of the dynamic concurrency of the system. While some services can consist in a simple procedure call, the mapping to different memory spaces (and thus to different processes) necessarily involves the inference of dynamic threads to service the calls.

By default, a default semantics for the communications is inferred for the port-to-port connectors, unless an explicit semantics is captured as was shown in Sect. 5.4.3. The methodology supports the capture and annotation of additional information which is specifically required and exploited by implementation tasks. Figure 5.22 illustrates how a channel instance can be annotated with information relevant for the synthesis of communications. Specifically, two additional attributes of the *«channel»* stereotype introduced in Sect. 5.4.3 enable to specify the communication stack and specific communication service employed. This specification capabilities were exploited in [37] to show how the different communication alternatives could be rapidly implemented and its impact in performance explored.

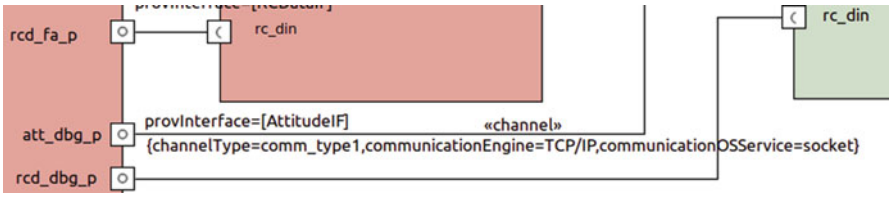


Fig. 5.22 Channel attributes exploited in software synthesis

Specific information can be also captured at lower levels for the SW synthesis phase. It was also mentioned that the *type* attribute of the *«OS»* stereotype is used to identify a specific target OS or RTOS distribution. In addition, the *«OS»* stereotype supports the capture of a set of drivers. This information is used to build up the so-called Board Support Package (BSP). At the HW platform modeling level, the processor Instruction-Set Architecture (ISA) is specified through the MARTE *«HwISA»* stereotype. ISA information is used for the selection of the suitable target at the compilation phase of the synthesis flow.

5.4.9 Verification Environment

The modeling methodology enables the capture of an environment model. This environment model is required by simulation-based performance analysis in order to describe the external stimuli and collect the functional outputs of the system.

The environment model is separately captured in the verification view, i.e., enclosed in a UML package decorated with the *«VerificationView»* stereotype. The environment model has a top component which instances the system components and an arbitrary number of environment components connected to the system component. Figure 5.23 shows the top environment component (*TopEnvQuadcopter*) and the environment components (in gray). The environment model is at a functional level, thus at the same abstraction level as the PIM. Therefore, it has to have client-server ports compatible with the system component, as illustrated in Fig. 5.23. Similarly as for the PIM, functionality is associated to the environment by means of artifacts and path constraints, thus using modeling techniques familiar to the system modeler. The functionality of environment components can invoke libraries and tools which facilitate the environment modeling. In performance model generation context, this functionality is not annotated. In a SW synthesis context, this functionality embeds code dealing with environment, i.e., target-dependent code accessing drivers.

For the identification of the top environment component and of the environment components, the methodology relies on the Universal Testing Profile (UTP), an OMG standard. Figure 5.24 shows the components of the verification view of the quadcopter.

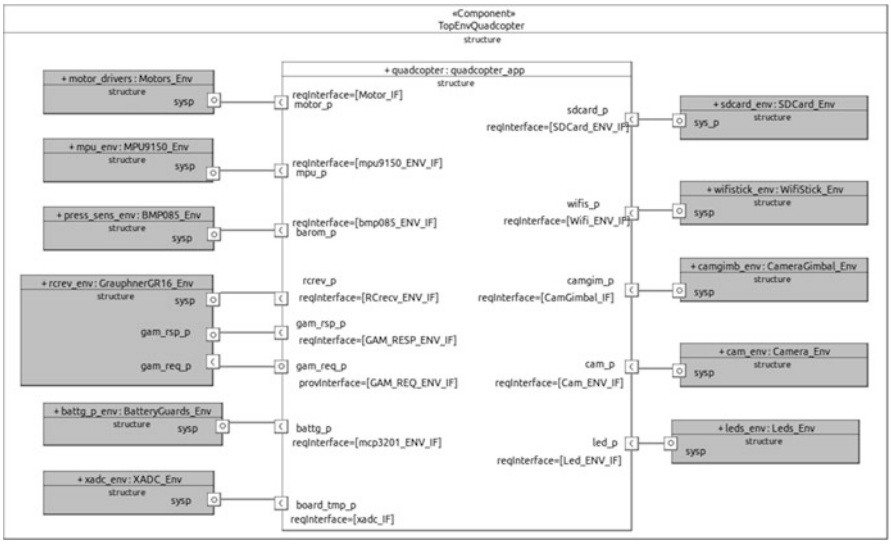


Fig. 5.23 The top environment component of the quadcopter instantiates environment components and connects them to the instance of the system component

5.4.10 Mixed-Criticality

The proposed modeling methodology enables the association of different criticalities to several types of modeling elements, including value annotations of non-functional types, constraints on extra-functional properties, application components, and platform resource components. These techniques rely on two minor extensions of the MARTE profile shown in Fig. 5.25. These extensions were required to support the CONTREX metamodel [25] (▶ Chap. 32, "Metamodeling and Code Generation in the Hardware/Software Interface Domain" introduces metamodeling), capable to cover the MCS modeling. The first extension adds a criticality attribute to a NFP constraint (left-hand side of Fig. 5.25). The criticality attribute is of integer type, which denotes an abstract criticality level. The NFP constraint can be associated then directly to different types of modeling elements, e.g., UML components and UML constraints. Therefore, this extension enables the association of criticalities to components, e.g., application and platform components, and also to constraints, employed to capture performance requirements and contracts. The second extension consists in enabling the annotation of a criticality value on a value annotation (right-hand side of Fig. 5.25). Again, the criticality value is an integer, denoting the criticality level.

These extensions support several modeling scenarios requiring mixed-criticality modeling. Figure 5.26 illustrates the direct association of criticalities to PIM component instances of the quadcopter. The association is performed through a *«NfpConstraint»* with its criticality value annotated in the *criticality* attribute. Please

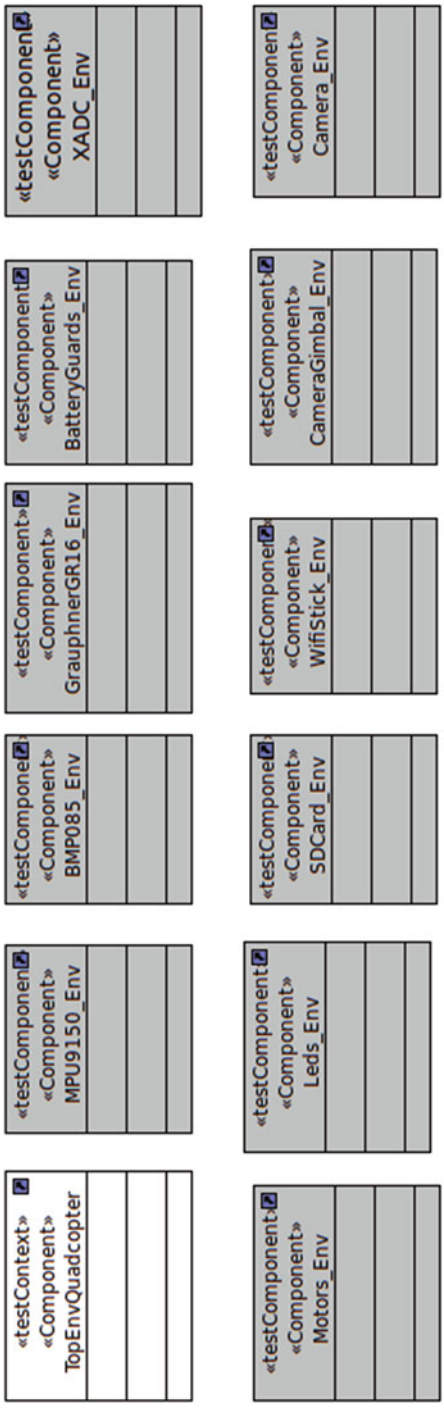


Fig. 5.24 Components of the environment model of the quadcopter

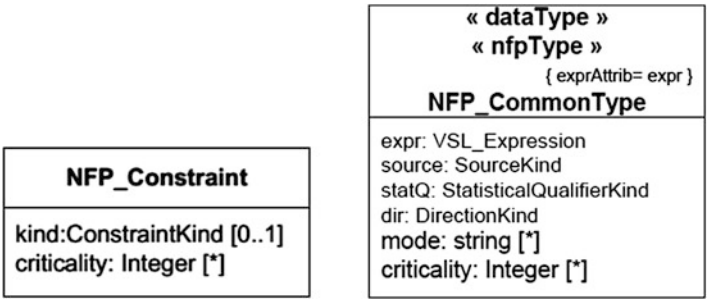


Fig. 5.25 Proposed MARTE extensions for mixed-criticality modeling

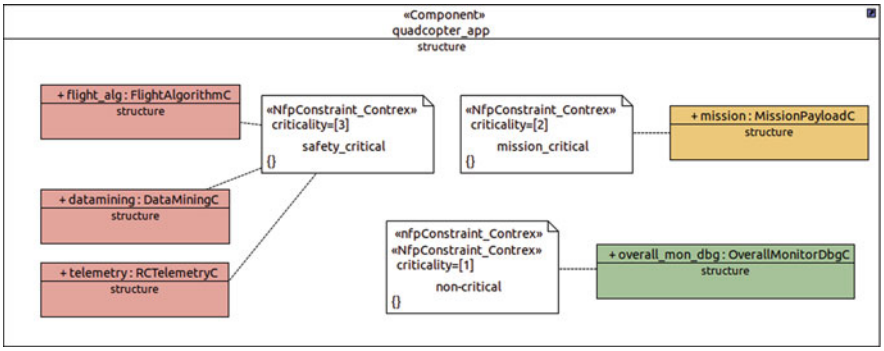


Fig. 5.26 Criticalities directly associated to PIM component instances

observe that Figure 5.26 actually shows the *«NfpConstraint_Contrex»* stereotype. This is provided by the methodology-specific eSSYN profile, and it is used as long as the aforementioned MARTE extension remains as a proposal. Similarly, criticalities can be also directly associated to other application components, e.g., memory spaces, and to software and platform components. Figure 5.27 shows the association of a criticality level to the computational resources of the quadcopter platform. The criticality associated to the application and platform component instances can then be used according to the design context. A main application of this direct association of criticality to PIM and platform components is the application of mapping rules at different levels (from application component to memory spaces, from application level to platform level), oriented to ensure a given degree of separation of resources. Other scenarios covered by this technique is when the development process of the components, either application or platform components, is conditioned by the criticality level, e.g., higher criticality components require more testbenches and more strict coding rules.

The methodology enables the association of criticalities to requirements and specifically to performance requirements. The methodology supports the association of criticalities to extra-functional requirements in several ways. An implicit

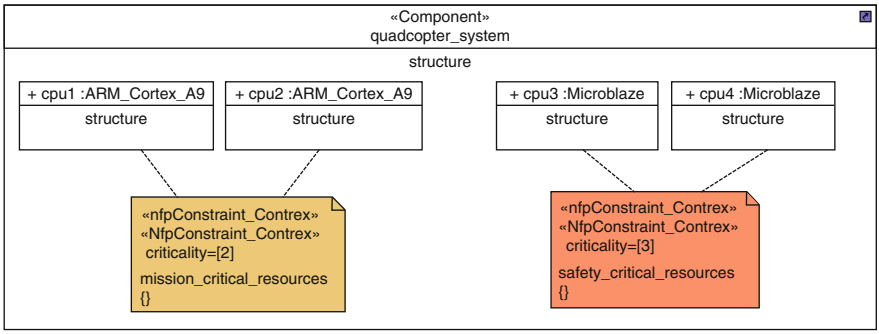


Fig. 5.27 Criticalities directly associated to HW computational elements

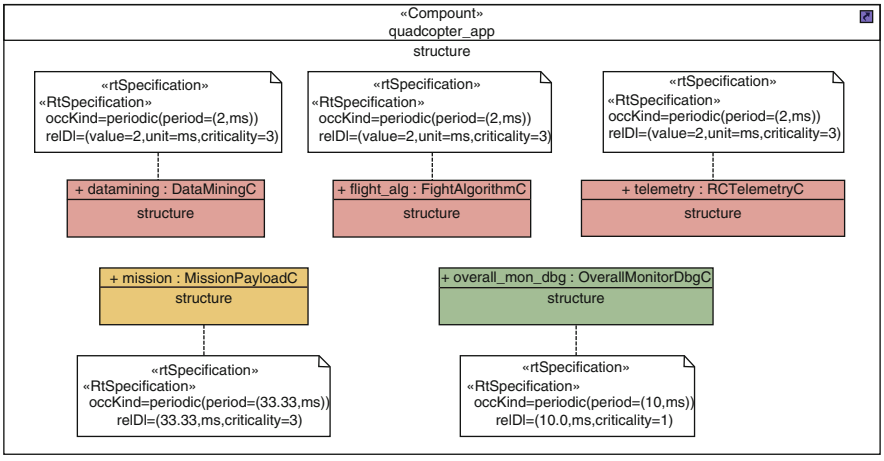


Fig. 5.28 Criticalities associated to deadline constraints

association is supported through the previously shown direct association of criticalities to components. In such a case, the methodology assumes that all the performance constraints associated to a component with an associated criticality inherit such a criticality. Additional techniques for associating criticalities to specific requirements are available in case a component has more than one associated requirement. In the constructs where the requirement on the extra-functional property is captured as a value annotation by means of VSL expression, now it is possible to annotate a criticality associated to the value annotation. Figure 5.28 reflects the association of the deadline requirements on the periodic tasks of the quadcopter and the criticalities annotations on the VSL expression; such a criticality value is associated to each specific deadline requirement.

When the performance requirement is expressed by means of a *«NfpConstraint»*, e.g, as it was the case in Fig. 5.17, the extension of the *«NfpConstraint»* with the criticality attribute can be used. Figure 5.29 shows the extension of the model

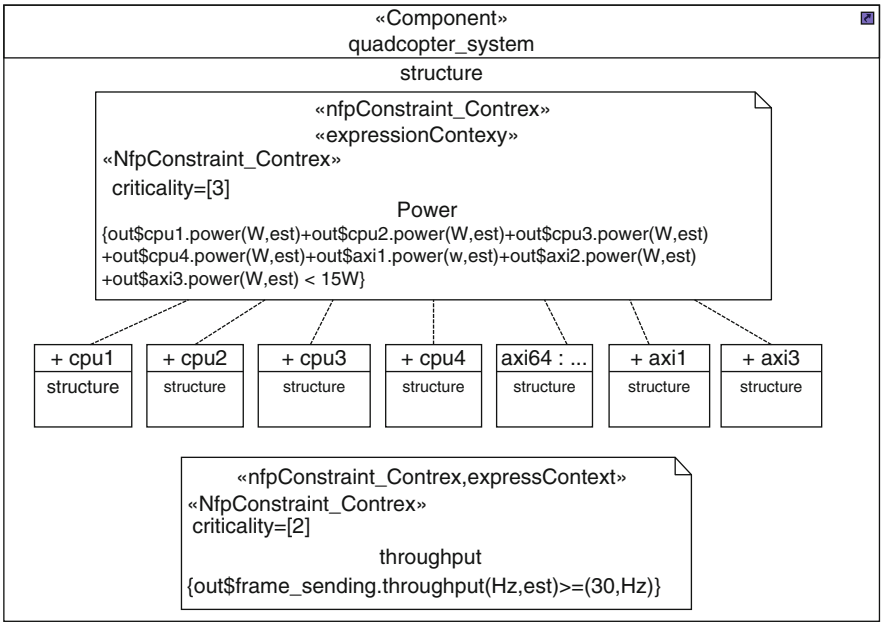


Fig. 5.29 Criticalities associated to system performance constraints

shown in Fig. 5.17, in order to specify the power dissipation constraint as safety critical (criticality = 3). This is because the heating can have side effects on the hardware executing the flight algorithm. The same technique is used to state that the throughput requirement associated to the PIM *mission* component (in charge to capture video, to detect and track an object, and to streaming the recorded action) is mission critical (criticality = 2).

Figure 5.30 shows another modeling scenario requiring mixed-criticality annotation. Schedulability theory on mixed-criticality systems rely on input models where a set of WCETs, instead of a single one, are associated to a task. Each WCET of the WCET set is associated to a criticality level. The proposed methodology covers this modeling need. Figure 5.30 illustrates the annotation of different worst-case execution times to the main application component instance. In the methodology, the implicit semantics associates the WCET to the main functionality of the component. The annotation relies on the *execTime* property of the *«ResourceUsage»* MARTE stereotype. The methodology assumes that a time annotation is associated not only to the application functionality but also to the computational resource which will run the workload. That is, the time annotation reflects the time taken by the execution of the application functionality in isolation conditions. Therefore, the annotation is performed through an association between the application component and the used computational component of the HW resource view. Specifically, such an association is a UML use dependency decorated with the MARTE *«ResourceUsage»* stereotype. The *execTime* property of the *«ResourceUsage»* is typed as a set of

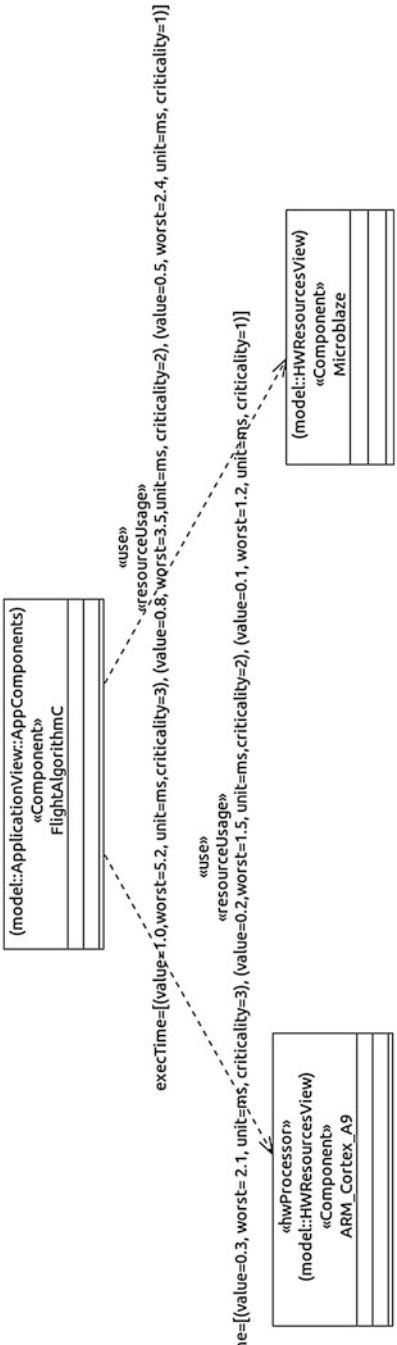


Fig. 5.30 Criticalities associated to different WCET value annotations

values of MARTE NFP_Duration type. Therefore, the *execTime* property can be and is used for annotating several values of execution time.

In addition, the methodology supports several *«ResourceUsage»* associations. This way, a single-source model provides information to trigger several schedulability analyses, e.g., for different mapping alternatives. Specifically, Fig. 5.30 states that if the flight algorithm component is mapped to a *microblaze* processor instance, then the WCETs to be considered are 5.2, 3.5, and 2.4 ms for criticalities 3, 2, and 1, respectively. Notice that this technique can be used only if a single functionality is associated to the component.

5.4.11 Modeling for Schedulability Analysis

This section shows the modeling elements and techniques used to do the validation of timing properties by means of schedulability analysis. The goal is to ensure real-time constraints (hence timing predictability) for critical tasks and evaluate the unused processing capacity that can be used for other non-critical activities in a simple way. This kind of real-time analysis is necessary when the systems are highly complex and have critical time constraints. The modeling for schedulability analysis is integrated in this component-based modeling methodology. The functionality is broken down into the internal and provided functions of communicating components, which in turn are mapped to the processing platform and where the analysis of many real-time situations is possible. A real-time situation corresponds to a workload and a platform. A workload is a set of end-to-end flows and their related stimuli (each end-to-end flow has its associated stimuli). An end-to-end flow describes the causal flow of actions that is triggered by external events and for which a deadline is specified. In such an end-to-end flow or execution path, one or more functions of one or more components can be executed. The objective of the analysis is to ensure that the response time of all the sequences of functions executed on each flow is smaller than the deadline associated.

For each individual function, the annotation of the worst-case and best-case execution times (WCET/BCET) is required. WCETs are needed to get upper bounds for response times and so to verify that the real-time requirements are met. The knowledge of BCETs is useful to reduce jitter and minimize pessimism in upper bounds for distributed systems [31]. The WCET and BCET of a given piece of functionality depends on the type of processing resource executing it. The mode of operation, and specifically, the operation frequency has also direct impact on the execution time of the same code segment. Section 5.4.10 showed how to annotate in the UML model different WCETs for different processor types. The operation modes are captured by means of UML state machines, where the operation modes are represented as UML states specified by the *«Mode»* MARTE stereotype. Transitions are represented as "UML transitions", specified by the *«ModeTransition»* MARTE stereotype.

In general, WCET annotations for independent functions are needed for performing schedulability analysis. However, the schedulability analysis introduced

before requires more information, in particular, the workload, the stimuli patterns that triggers the workload, the platform where it runs, and the hard real-time requirements. Moreover, the modeling methodology shall enable the specification of the ambit and elements of the model involved in the real-time analysis, i.e., the functions and resources of which are really involved out of the overall model.

The schedulability analysis model is contained in a specific view package, stereotyped as *«SchedulabilityView»*. A real-time situation corresponding to an execution model is included in this package and consists in one or more end-to-end flows. The end-to-end flow is modeled as a UML activity decorated with the *«SaEndtoEndFlow»* MARTE stereotype. An activity diagram is used to describe the causal flow as a sequence of *steps*. In general, a *step* represents the usage of resources needed at that point in the flow. Here, it is used to express the execution time taken from the processing resource associated and the messages sizes that are to be sent through network channels. The characteristics of the event that triggers the end-to-end flow are annotated in the initial node of the activity of the end-to-end flow by means of the *«GaWorkLoadEvent»* MARTE stereotype. This stereotype allows to indicate the pattern of activation (e.g., periodic or sporadic) and its parameters (e.g., period). Each step in the flow is modeled in the activity diagram as a UML opaque action annotated with the *«SaStep»* MARTE stereotype. The attribute *execTime* of *«SaStep»* is used to capture the worst and the best execution times of the function involved. The steps may be specified at different granularity levels depending on the knowledge available for the underlying execution elements. At the finest granularity, steps are used on functions with known WCET/BCET. A step can also be used to model operations that invoke other steps. The *concurRes* attribute of a *step* is used to indicate the task it runs on, in the case of a computational step, or the channel through which the message is sent, in the case of a communication step. In the former case, a computation step may also model a sequence of operations executed by the task by means of the *subusage* attribute. Each operation in the subusage sequence is modeled statically by means of UML operations stereotyped with *«SaStep»*. Figure 5.31 shows (on the left-hand side) the modeling of two end-to-end flows for the quadcopter case: one for the data miner task and one for the flight algorithm task. The right-hand side of Fig. 5.31 shows the activity diagram of one of the end-to-end flows, which illustrates the simplest way to describe it.

Schedulability analysis requires to explicitly model the concurrent *tasks* the aforementioned *steps* belong to, i.e., *steps* are mapped to those tasks through the *concurRes* attribute. These tasks are specified through UML properties with the *«SchedulableResource»* MARTE stereotype in the concurrency view. Figure 5.32 shows the modeling of the quadcopter tasks (one for the datamining and another one for the flight algorithm) involved in the schedulability analysis.

In turn, these tasks are associated to the HW/SW platform resources that will execute them by means of the *host* attribute of the *«SchedulableResource»* stereotype. The *«SaExecHost»* and *«SaCommHost»* MARTE stereotypes are employed for indicating the platform resources involved in the schedulability analysis. Specifically, the *«SaExecHost»* stereotype indicates platform component instances where

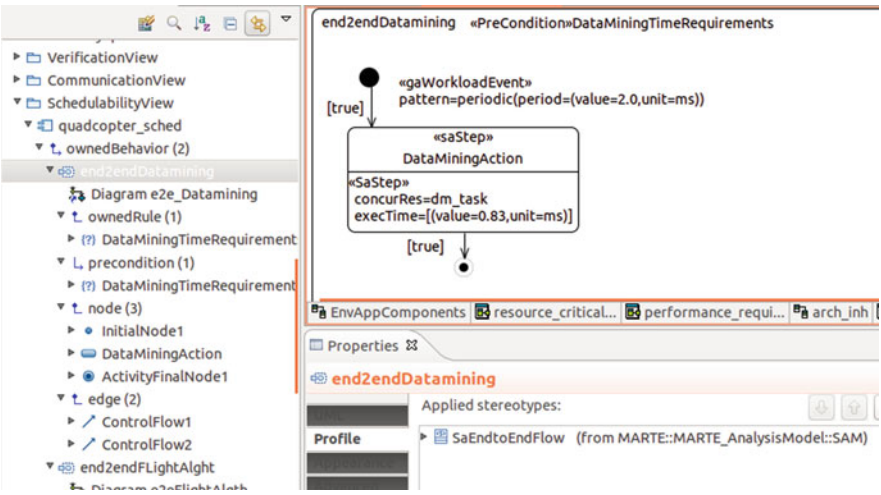


Fig. 5.31 Specification of end-to-end flows in the quadcopter

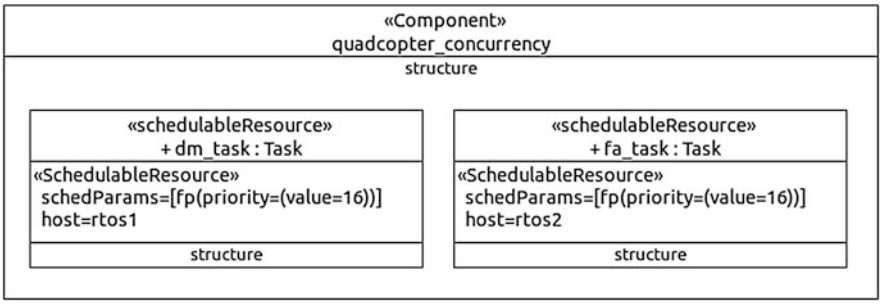


Fig. 5.32 Modeling of quadcopter tasks and their mapping to the SW/HW platform for schedulability analysis

tasks can be scheduled and thus mapped and enables capturing properties such as the range of priorities and context switching times. This way, each *host* attribute of the quadcopter tasks involved in schedulability analysis, shown in Fig. 5.32, can only point to any of the platform resources stereotyped as *«SaExecHost»* in the architectural view of the quadcopter, that is, *rtos1* and *rtos2*, shown in Fig. 5.33.

5.5 Single-Source Design Framework

The single-source modeling methodology introduced in this chapter is documented in detail in [42]. The modeling methodology enables the production of models which serve as an input to a tool infrastructure supporting a single-source design approach. The *CONTREX Eclipse plug-in* (*CONTREP*) [41] is the unified, graphical front-end of that infrastructure. As shown in Fig. 5.34, for the modeling activity,

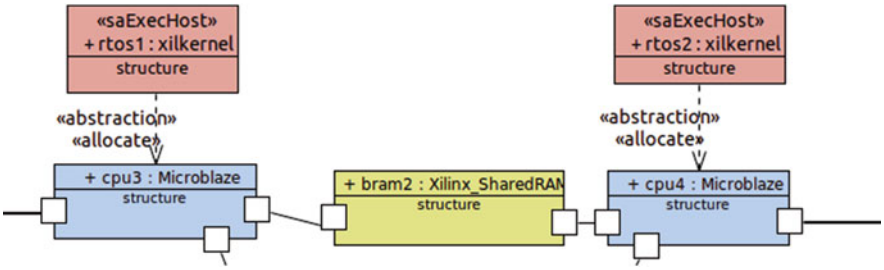


Fig. 5.33 Indication of the execution resources in the SW/HW platform architecture of the quadcopter for schedulability analysis

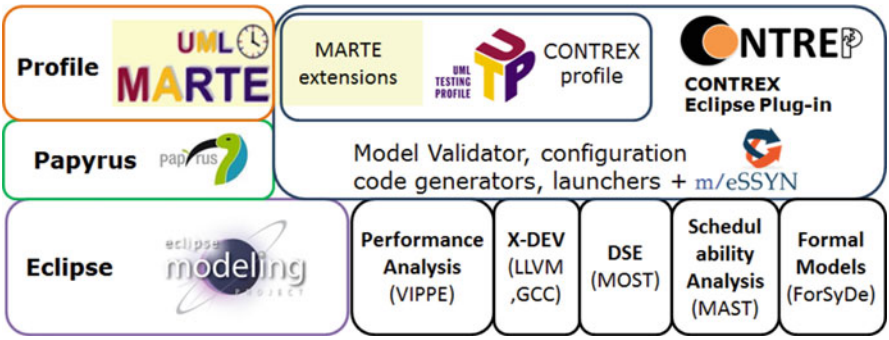


Fig. 5.34 The CONTREX Eclipse plug-in takes the UML/MARTE model as an input for the design activities which rely on different tools

CONTREP provides the eSSYN profile, which provides an implementation of the UTP stereotypes and of a minimum set of methodology-specific extensions employed by the modeling methodology. In addition, CONTREP also integrates a validation tool for the detection of the violation of the modeling rules. As Fig. 5.35 reflects, as well as modeling, CONTREP enables further design activities. For room reasons, it is not possible to illustrate here the application of a complete use case. However, related publications have reported how the UML model served for the generation of a functional model and for SW synthesis [37] relying on eSSYN. For schedulability analysis, the MARTE2MAST generator [10] has been adapted for its integration in CONTREP, and so to enable the automatic generation of the input for the MAST tool [11]. Schedulability analysis requires also tools for obtaining the WCETs and BCETs. In [34], simulation was used to obtain maximum observed times as an early approach. The framework is also capable to generate a ForSyDe executable model [14] (see ▸ Chap. 4, "ForSyDe: System Design Using a Functional Language and Models of Computation") for formal functional validation. CONTREP also automates the generation of a fast performance model. The performance model relies on the VIPPE tool [43], which implements advanced techniques for fast simulation and performance assessment. Simulation is convenient for getting accuracy and considering the dynamism of the application and of the input stimuli

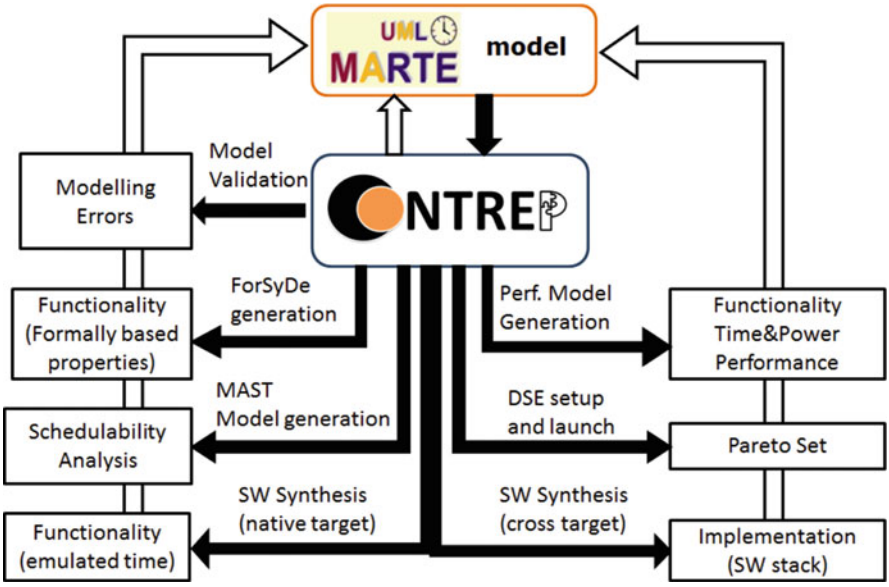


Fig. 5.35 Design activities that can be launched with the CONTREX Eclipse plug-in

of the system environment, that is, for an scenario-aware assessment of the system (▸ Chap. 9, "Scenario-Based Design Space Exploration"). Specifically, VIPPE relies on host-compiled simulation (see ▸ Chap. 19, "Host-Compiled Simulation") and is capable to parallelize the simulation (▸ Chap. 17, "Parallel Simulation" introduces advanced parallelization techniques in a SystemC context) to exploit multi-core host platforms.

CONTREP also automates the generation of a complete simulation-based DSE infrastructure. As well as the performance model, files describing information like the design space, the performance constraints, the cost functions, and the exploration strategy, a basic input for the exploration tool coupled to the simulatable performance model is generated. The framework is flexible and allows the user to select from the CONTREP front-end, at a high abstraction level, among different exploration strategies (see ▸ Chap. 6, "Optimization Strategies in Design Space Exploration"), exploration tools, and report options. Moreover, the framework enables also the launch of the automated DSE process and the visualization of results from the own graphical environment.

5.6 Conclusions

This chapter has presented the main modeling techniques of a single-source modeling methodology relying on the UML language and the MARTE profile, both OMG standards. These techniques enable the methodology to support separation of

concerns, incremental modeling, and functional component modeling and to feed ESL key design tasks such as design space exploration and software synthesis. All these capabilities are mandatory for the productivity boost required by the current complexities of modern embedded systems.

More information on the introduced methodology can be found in related publications [12–16, 34, 37], in the methodology related website [44], and in the CONTREX website [28].

The presented methodology maximizes the exploitation of UML and specifically of the MARTE and UTP profiles for complex embedded system modeling. This does not necessarily mean that these profiles are currently capable to cover all the required concepts. In most of the cases, the methodology covers the lacks by proposing minimal extensions to MARTE, for instance, for the annotation of criticalities, for the description of a design space in VSL, or for the capture of the system views. In other cases, more important extensions are required. This is the case of embedded distributed systems description, for which CONTREX has made also a proposal [9, 17].

Acknowledgments This chapter has been partially funded by the European FP7 611146 (CONTREX) project and by the Spanish TEC 2014-58036-C4-3-R (REBECCA) project. We are thankful to the OFFIS team in CONTREX for their support and for all the documentation and material on their quadcopter implementation. This includes the quadcopter picture of the chapter.

References

1. Alam O, Kienzle J (2013) Incremental software design modelling. In: Proceedings of the 2013 conference of the center for advanced studies on collaborative research, CASCON '13. IBM Corp., Riverton, pp 325–339
2. Ambler SW (2015) Single source information: an Agile best practice for effective documentation. http://agilemodeling.com/essays/singleSourceInformation.htm
3. Arpinen T, Salminen E, Hämäläinen TD, Hännikäinen M (2012) {MARTE} profile extension for modeling dynamic power management of embedded systems. J Syst Archit 58(5):209–219. doi:10.1016/j.sysarc.2011.01.003. Model Based Engineering for Embedded Systems Design
4. Bailey B, Martin G, Piziali A (2007) ESL design and verification: a prescription for electronic system level methodology. Morgan Kaufmann/Elsevier, Amsterdam/Boston
5. Bakshi A, Prasanna VK, Ledeczi A (2001) Milan: a model based integrated simulation framework for design of embedded systems. In: Proceedings of the 2001 ACM SIGPLAN workshop on optimization of middleware and distributed systems, OM '01. ACM, New York, pp 82–93. doi:10.1145/384198.384210
6. Burns A, Davis R (2015) Mixed-criticality systems: a review, 6th edn. Technical report, Department of Computer Science, University of York
7. Cabot J (2014) Single-source modeling for embedded systems with UML/MARTE. http://modeling-languages.com/modeling-embedded-systems-uml-marte
8. Dekeyser J, Gamatie A, Atitallah R, Boulet P (2008) Using the UML profile for MARTE to MPSoC co-design. In: 1st international conference on embedded systems and critical applications (ICESCA'08)
9. Ebeid E, Medina J, Quaglia D, Fummi F (2015) Extensions to the UML profile for MARTE for distributed embedded systems. In: 2015 forum on specification and design languages (FDL), pp 1–8

10. Garcia A, Medina J: MARTE2MAST. http://mast.unican.es/umlmast/marte2mast/
11. Gonzalez M, Gutierrez JJ, Palencia JC, Drake JM (2001) Mast: modeling and analysis suite for real time applications. In: 2001 13th Euromicro conference on real-time systems, pp 125–134. doi:10.1109/EMRTS.2001.934015
12. Herrera F, Penil P, Posadas H, Villar E (2014) Model-driven methodology for the development of multi-level executable environments. In: Haase J (ed) Models, methods, and tools for complex chip design. Lecture notes in electrical engineering, vol 265. Springer International Publishing, pp 145–164. doi:10.1007/978-3-319-01418-0_9
13. Herrera F, Posadas H, Peñil P, Villar E, Ferrero F, Valencia R, Palermo G (2014) The COMPLEX methodology for UML/MARTE modeling and design space exploration of embedded systems. J Syst Archit 60(1):55–78. doi:10.1016/j.sysarc.2013.10.003
14. Herrera F, Peñil P, Villar E (2015) Enhancing analyzability and time predictability in UML/MARTE component-based application models. In: Proceedings of the 18th international workshop on software and compilers for embedded systems, FDL '15. IEEE
15. Herrera F, Peñil P, Villar E (2015) A model-based, single-source approach to design-space exploration and synthesis of mixed-criticality systems. In: Proceedings of the 18th international workshop on software and compilers for embedded systems, SCOPES '15. ACM, New York, pp 88–91. doi:10.1145/2764967.2784777
16. Herrera F, Peñil P, Villar E (2015) UML/MARTE modelling for design space exploration of mixed-criticality systems on top of predictable platforms. In: Jornadas de Computación Empotrada (JCE'15)
17. Herrera F, Peñil P, Villar E (2015) Extension of the UML/MARTE network modelling methodology in CONTREX. Technical report, University of Cantabria. http://umlmarte.teisa.unican.es/wp-content/uploads/2016/05/ExtendedUML_MARTE_Network_Modelling_Methodology.pdf
18. Intel (2014) Intel cofluent methodology for SysML: UML*SysML*MARTE flow for Intel CoFluent studio. http://www.intel.com/content/www/us/en/cofluent/cofluent-methodology-for-sysml-white-paper.html
19. ITRS: International roadmap of semiconductors. http://www.itrs.net/
20. Kang E, Jackson E, Schulte W (2011) An approach for effective design space exploration. In: Calinescu R, Jackson E (eds) Foundations of computer software. Modeling, development, and verification of adaptive systems. Lecture notes in computer science, vol 6662. Springer, Berlin/Heidelberg, pp 33–54. doi:10.1007/978-3-642-21292-5_3
21. Kangas T, Kukkala P, Orsila H, Salminen E, Hännikäinen M, Hämäläinen TD, Riihimäki J, Kuusilinna K (2006) UML-based multiprocessor soc design framework. ACM Trans Embed Comput Syst 5(2):281–320. doi:10.1145/1151074.1151077
22. Kruchten P (1995) Architecture blueprints—the "4+1" view model of software architecture. In: Tutorial proceedings on TRI-Ada '91: Ada's role in global markets: solutions for a changing complex world, TRI-Ada '95. ACM, New York, pp 540–555. doi:10.1145/216591.216611
23. Lemke M (2012) Mixed criticality systems. Report from the workshop on mixed criticality systems. Technical report, Information Society and Media Directorate-General
24. Liehr AE (2009) Languages for embedded systems and their applications. Lecture notes in electrical engineering, vol 36. Springer Netherlands, pp 43–56. doi:10.1007/978-1-4020-9714-0_3
25. Medina J et al (2015) CONTREX system metamodel. Technical report. https://contrex.offis.de/home/images/publicdeliverables/Deliverable%20D2.1.1%20v1.0.pdf
26. Mura M, Murillo L, Prevostini M (2008) Model-based design space exploration for RTES with SysML and MARTE. In: Forum on specification, verification and design languages, FDL 2008, pp 203–208. doi:10.1109/FDL.2008.4641446
27. Nicolescu G, Mosterman P (2009) Model-based design for embedded systems. Computational analysis, synthesis, and design of dynamic systems. CRC Press, Boca Raton
28. OFFIS (2015) CONTREX FP7 project website. https://contrex.offis.de/home/
29. OMG (2011) OMG UML profile for MARTE, modelling and analysis of real-time embedded systems, Version 1.1. Available at www.omg.org

30. OMG (2015) OMG unified modeling language. Available at www.omg.org
31. Palencia JC, Gutierrez JJ, Gonzalez Harbour M (1998) Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems. In: Proceedings of the 10th Euromicro workshop on real-time systems, pp 35–44. doi:10.1109/EMWRTS.1998.684945
32. Panunzio M, Vardanega T (2009) On component-based development and high-integrity real-time systems. In: IEEE 19th international conference on embedded and real-time computing systems and applications
33. Peñil P, Posadas H, Nicolás A, Villar E (2012) Automatic synthesis from UML/MARTE models using channel semantics. In: Proceedings of the 5th international workshop on model based architecting and construction of embedded systems, ACES-MB '12. ACM, New York, pp 49–54. doi:10.1145/2432631.2432640
34. Peñil P, Posadas H, Medina J, Villar E (2015) UML-based single-source approach for evaluation and optimization of mixed-critical embedded systems. In: DCIS'15
35. Piel E, Atitallah RB, Marquet P, Meftali S, Niar S, Etien A, Dekeyser J, Boulet P (2008) Gaspard2: from MARTE to SystemC simulation. In: Design, automation and test in Europe (DATE 08)
36. Pop A, Akhvlediani D, Fritzson P (2007) Integrated UML and modelica system modeling with modelicaml in Eclipse. In: Proceedings of the 11th IASTED international conference on software engineering and applications, SEA '07. ACTA Press, Anaheim, pp 557–563
37. Posadas H, Peñil P, Nicolás A, Villar E (2014) Automatic synthesis of embedded {SW} for evaluating physical implementation alternatives from UML/MARTE models supporting memory space separation. Microelectron J 45(10):1281–1291. doi:10.1016/j.mejo.2013.11.003. DCIS'12 Special Issue
38. Szyperski C (2002) Component software: beyond object-oriented programming. Addison Wesley, London
39. TILLOO R (2015) What is incremental model in software engineering? It's advantages and disadvantages. Available in http://www.technotrice.com/incremental-model-in-software-engineering
40. Truyen F (2006) The fast guide to model driven architecture – the basics of model driven architecture. http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf
41. University of Cantabria: CONTREX Eclipse plug-in website. http://contrep.teisa.unican.es. Accessed 04 Oct 2016
42. University of Cantabria: UML/MARTE single-source methodology website. http://umlmarte.teisa.unican.es. Accessed: 04 Oct 2016
43. University of Cantabria: VIPPE website. http://vippe.teisa.unican.es. Accessed 04 Oct 2016
44. University of Cantabria (2016) eSSYN website. http://eSSYN.com
45. Vidal J, de Lamotte F, Gogniat G, Soulard P, Diguet JP (2009) A co-design approach for embedded system modeling and code generation with UML and MARTE. In: Design, automation test in Europe conference exhibition. DATE '09, pp 226–231. doi:10.1109/DATE.2009.5090662
46. Woods E, Rozanski N (2005) Using architectural perspectives. In: 2014 IEEE/IFIP conference on software architecture, vol 0, pp 25–35. doi:10.1109/WICSA.2005.74

Part III
Design Space Exploration

Optimization Strategies in Design Space Exploration

6

Jacopo Panerati, Donatella Sciuto, and Giovanni Beltrame

Abstract

This chapter presents guidelines to choose an appropriate exploration algorithm, based on the properties of the design space under consideration. The chapter describes and compares a selection of well-established multi-objective exploration algorithms for high-level design that appeared in recent scientific literature. These include heuristic, evolutionary, and statistical methods. The algorithms are divided into four sub-classes and compared by means of several metrics: their setup effort, convergence rate, scalability, and performance of the optimization. The common goal of these algorithms is the optimization of a multi-processor platform running a set of diverse software benchmark applications. Results show how the metrics can be related to the properties of a target design space (size, number of variables, and variable ranges) with a focus on accuracy, precision, and performance.

Acronyms

ADRS	Average Distance from Reference Set
ANN	Artificial Neural Network
DoE	Design of Experiments
DSE	Design Space Exploration
EA	Evolutionary Algorithm
GA	Genetic Algorithm
ILP	Integer Linear Program
MDP	Markov Decision Process
MPSoC	Multi-Processor System-on-Chip
NN	Neural Network

J. Panerati (✉) • G. Beltrame
Polytechnique Montréal, Montreal, QC, Canada
e-mail: jacopo.panerati@polymtl.ca; giovanni.beltrame@polymtl.ca

D. Sciuto
Politecnico di Milano, Milano, Italy
e-mail: donatella.sciuto@polimi.it

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_7

PSO	Particle Swarm Optimization
RSM	Response Surface Modeling
SA	Simulated Annealing
SoC	System-on-Chip

Contents

6.1 Introduction

Given a specific software application – or a class of software Applications – the parameters of a System-on-Chip (SoC) can be appropriately tuned to find the best trade-offs among the figures of merit (e.g., energy, area, and delay) deemed of interest by the designer. Design Space Exploration (DSE) is the tool by which the optimal *configuration* for a given system can be found.

This parameter tuning is fundamentally an optimization problem, which generally involves the maximization (or minimization) of multiple objectives. A consequence of having multiple objectives is that optimal solutions are, potentially, no longer unique. A set of *metrics* or *objective functions* are used to express the quality of a solution from different perspectives, also known as objectives. Since multi-objective optimization problems do not have a single optimal solution, solutions consist instead of several optima, i.e., the points that lie on the *Pareto curve* [11]. These are the optimal points that are non-dominated by any other point.

To find the Pareto curve for a specific platform, the designer has to evaluate all the possible configurations of the design space and characterize them in terms of objective functions. This approach is known as full or exhaustive search, and it is

often impractical due to the large number of points in a design space and/or due to the high cost associated with the evaluation of the objective functions (e.g., long simulation times).

Even today, Multi-Processor Systems-on-Chips (MPSoCs) platforms are often tuned according to designer experience or the non-systematic application of several algorithms found in literature. For example, classical heuristic algorithms (i.e., tabu search, simulated annealing, etc.) [28] are extremely common, as well as techniques able to reduce the design space size [22]. The advanced hybrid approaches in ▸ Chap. 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration" combine metaheuristics and search algorithms to avoid large, unfeasible areas of the design space. All these approaches need to use simulation (or estimation) to assess the system-level metrics (i.e., the objective functions) of the very large number of configurations they evaluate. For a review of microarchitecture-level modeling and design methodologies for SoC, please refer to ▸ Chap. 27, "Microarchitecture-Level SoC Design".

The characteristics of a design space (e.g., its size or the time required to simulate one of its points) can certainly affect how different exploration algorithms perform in terms of time to convergence or accuracy of results. This chapter describes and compares 15 algorithms among the several ones that have been recently proposed in the scientific literature for automatic DSE and multi-objective optimization. These different approaches are dissimilar in terms of theoretical background, applicability conditions, and overall performance. Through rigorous analysis, the advantages and drawbacks of each method are uncovered, providing guidelines for their use in different contexts.

In the following, Sect. 6.2 describes a partitioning of the existing literature into four classes; Sect. 6.3 reviews the methodology of each one of the 15 algorithms that underwent experimental comparison; Sect. 6.4 presents the experimental setup, the framework used to compare these algorithms, and its results; Sect. 6.5 contains the discussion of these results as well as recommendations on the use of the algorithms; Sect. 6.6 lists some of the currently available open-source implementations of the algorithms; finally, Sect. 6.7 draws some concluding remarks.

6.2 Classification of Multi-objective DSE Strategies

The automation of DSE can be split into two sub-problems: (a) the identification of plausible candidate solutions (i.e., valid system configurations) and (b) the evaluation of the metrics of interest of these solutions, in order to select the optimal configurations.

Evolutionary Algorithms (EAs) in particular have widespread use in the area of design space exploration. EAs discriminate and select solutions using a combination of metrics called the fitness function. EAs are usually easy to apply and do not require detailed knowledge of the design space to be explored, and there is a strong theoretical background on how to assign fitness values for multi-objective problems [9].

To classify these methods, Coello [4] proposed to divide evolutionary approaches for multi-objective optimization into three sub-classes according to the way in which the fitness of individuals/solutions is computed: (1) algorithms using aggregating functions, (2) algorithms using non-aggregating but non Pareto-based approaches, and (3) Pareto-based approaches, such as the NSGA algorithm.

Broadening the scope of the analysis to also include design space exploration methods that are not based on EAs, four different classes of algorithms for problem (a) can be found in the literature:

- **Class 1: Heuristics and pseudo-random optimization approaches** attempt to reduce the design space under scrutiny and focus the exploration on regions of interest [10, 11]. Methodologies in this class often rely on full search or pseudo-random algorithms to explore the selected regions. Class 1 includes algorithms such as multi-agent optimization (e.g., Particle Swarm Optimization (PSO) [26]), Simulated Annealing (SA), tabu search, and operations research algorithms. Methodologies in this class often aim at drastically reducing the number of configurations to evaluate (e.g., from the product to the sum of the number of tunable parameters in [10]). One way to do so is, for example, to identify sub-spaces (called clusters) into the design search space that can be efficiently explored in an exhaustive fashion [11]. The global Pareto front can then be reconstructed from the Pareto-optimal configurations of each partition. Because of their simplicity, the exploration results of the algorithms in this class remain in many cases sub-optimal.
- **Class 2: Evolutionary algorithms.** EAs are the most common and widely used DSE algorithms, they apply random changes of a starting set of configurations to iteratively improve their Pareto set of solutions. Genetic Algorithms (GAs) belong to this category [28]. The major advantages of these algorithms are the very limited setup effort and the fact that they do not require any specific knowledge associated to the search space or the metrics used for the optimization. Unfortunately, these algorithms are not guaranteed to find the optimal solutions or even to converge to results within certain predefined quality bounds. In practice, however, they usually perform fairly well when they are allowed to run for a sufficiently large number of evaluations. Techniques in this class can also be combined with exact methods. For example, in [20], DSE is translated into a multi-objective 0–1 Integer Linear Program (ILP) problem and a pseudo-Boolean (PB) solver is used to constrain a GA within the feasible search space.
- **Class 3: Statistical approaches without domain knowledge.** Methodologies in this class extract a *metamodel* from the design space and use it to predict the next configurations to evaluate [20, 21, 27, 32]. Class 3 includes those methods that use statistics to guide their DSE. Many of the algorithms in this class exploit a methodology called *Design of Experiments (DoE)* [21, 27, 32] to characterize the sensitivity of the system to its parameters. DoE, in fact, allows to estimate the portion of the variance of each objective metric associated to the oscillations in a certain parameter. Heuristics or metamodels are then developed from this sensitivity analysis and used to tune the parameters and find the optimal configurations of the system. The work in [27] is an example of this

sort of approach. It uses DoE to define an initial set of experiments and create a preliminary estimate of the target design space. Then, the exploration is further refined iteratively using a technique called Response Surface Modeling (RSM). Statistical methods, such as DoE, are the defining characteristics of this class, as they allow to extract the maximum amount of information from the initial training sets of limited size. The generated metamodels can then be used both to find new candidate configurations as well as to evaluate them.

- **Class 4: Statistical approaches with domain knowledge.** These are techniques that use predefined rules and knowledge specific to a certain design space to find the most promising solutions [3]. The algorithms in Class 4 are characterized by the use of built-in domain knowledge to set up a probabilistic framework to guide the identification of new candidate solutions. The work in [3], in particular, combines decision theory with this kind of integrated knowledge. The DSE problem is remapped to a Markov Decision Process (MDP) [18, 30] whose solution is the sequence of changes to apply to the tunable parameters in order to minimize (or maximize) one of the objective performance metrics. The major advantage of this approach is that it greatly reduces the number of times in which the system requires to be simulated (i.e., only when uncertainty at a fork is too large to be managed with the embedded domain knowledge).

Later in this chapter, a selection of well-established algorithms from all the four classes and their configuration parameters are described. Then, their strengths and weaknesses are compared in the context of the optimization of a symmetric multi-processor platform. The results of this experiment provide guidelines for the selection of the right DSE and multi-objective optimization algorithm, given the characteristics of the design space.

The evaluation of the candidate solutions – problem (b) – can be tackled either using detailed simulation [3] or simpler predictive models [14], or also a combination of the two.

Okabe, Jin, and Sendhoff [25] reviewed several metrics for the evaluation of solutions to multi-objective optimization problems. Their conclusion is that no single metric is usually sufficient to quantify the quality of the Pareto set returned by a multi-objective optimization algorithm. Similar results are also found in [35].

In our analysis, four different metrics are used to evaluate the accuracy, distribution, and cardinality of the Pareto sets found by the optimization algorithms. Moreover, the performance of the algorithms under study is quantified as the number of evaluations needed to converge to a Pareto set.

6.3 Multi-objective DSE Algorithms

This section details 15 multi-objective optimization algorithms for DSE that are then experimentally compared in Sect. 6.4. For each algorithm, a brief description of inner working and its parameters are given. The reader should pay attention to the fact that the parameters mentioned here are not the tunable parameters of a computing platform (i.e., those that create the design space). The parameters

described in this section are the configuration parameters of each algorithm and they are important as they influence its performance and setup effort.

6.3.1 Heuristics and Pseudo-random Optimization Approaches

This section describes algorithms belonging to the first class defined in Sect. 6.2 in detail. Half of the algorithms described in this section (MOSA, PSA, and SMOSA) are based on SA. SA is essentially a local search characterized by the way in which local minima are avoided: solutions that do not improve the current one can be accepted with a given probability p, computed from a Boltzmann distribution (parameterized by a coefficient T, called temperature).

6.3.1.1 Adaptive Windows Pareto Random Search (APRS)

The APRS algorithm is one of two novel algorithms implemented in the Multicube Explorer framework [37]. APRS takes an initial set of design points as its approximate Pareto set, then it improves this solution by randomly picking new points from windows centered on the current points. The size of the windows is reduced according to two factors: time (i.e., number of iterations) and quality of the points they are created from.

6.3.1.2 Multi-objective Multiple Start Local Search (MOMSLS)

MOMSLS [16] is a heuristic method based on executing in parallel multiple local searches starting each one of them from a different initial point. Each local search itself is a simpler heuristic that iteratively refines an initial random solution by looking for better candidates in its neighborhood [30]. As a consequence, MOMSLS evaluates solutions in N different neighborhoods during each one of its iteration steps.

6.3.1.3 Multi-objective Particle Swarm Optimization (MOPSO)

PSO [19] is a biologically inspired heuristic search method that mimics the movement of a flock of birds. Candidate solutions are described by particles of the swarm, and, at each iteration, they move a maximum of the objective function updating their position according to a velocity vector (see Fig. 6.1) that is the linear combination of three components:

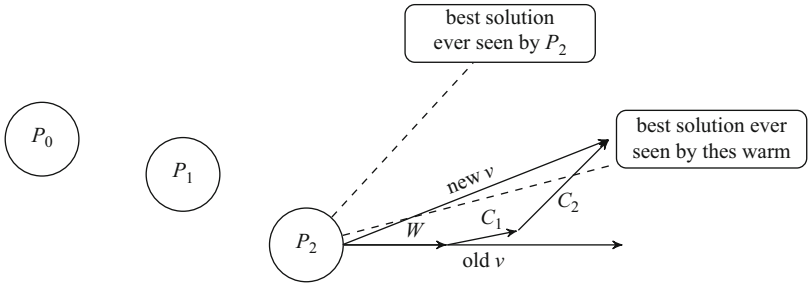


Fig. 6.1 Example of a velocity vector update for a particle belonging to a three-particle swarm

- their last velocity vector (weighted by an inertial factor, W);
- the direction toward the best (e.g., with the greatest objective metric) solution ever reached by the swarm (weighted by a social learning factor, C_1);
- the direction toward the best solution ever reached by the individual particle (weighted by a cognitive learning factor, C_2).

To adapt PSO to multi-objective optimization problems, MOPSO [26] replicates the PSO approach in N different swarms. These N swarms have, for objective function, the product of the multiple objective functions, each elevated by a random exponent. Moreover, in MOPSO, the inertial and social learning factors are set to 0. Particles are also forced to perform a random walk with a fixed probability p to avoid local minima.

In MOPSO implementations where the social learning factor is different from 0, the overall performance is also influenced by the way in which the swarm "leaders" are selected. The analysis performed in [23] shows that choosing as leaders those particles/solutions that contribute the most to the hyper-volume of the Pareto front usually results in the best performance.

6.3.1.4 Multi-objective Simulated Annealing (MOSA)

MOSA [36] is the multi-objective adaptation of the well-known SA technique. In [36], there are two proposed approaches for the translation of SA into the context of multi-objective optimization:

- probability scalarization, i.e., computing the acceptance/rejection probability of new solutions for each performance metric and their aggregation;
- criterion scalarization, i.e., the projection of the performance metrics into a single metric that is then used to compute the acceptance/rejection probability of the new solution.

The approach actually implemented in MOSA [36] and tested in Sect. 6.4 is the second one, while the first one is used by the two other SA-based algorithms presented in the following.

6.3.1.5 Pareto Simulated Annealing (PSA)

PSA [5] proposes two different criteria to scalarize the acceptance/rejection probability of a new solution and adapt simulated annealing to multi-objective problems:

- rule C says that the rejection probability p is proportional to the largest difference between the current and the new solution among all the performance metrics under evaluation;
- rule SL, instead, states that p is the weighted linear combination of the differences between the current and the new solution in all their performance metrics.

The weights used by rule SL to multiply each metric are also tuned at each iteration, depending on whether the most recently introduced solution brought a deterioration in that specific metric or not.

6.3.1.6 Serafini's Multiple Objective Simulated Annealing (SMOSA)

The work in [31] enriches the discussion on the rules that can be used to combine multiple performance metrics and apply SA to multi-objective optimization problems. In [31], a new complex rule is defined as the linear composition, with coefficients α and $(1-\alpha)$, of two simpler rules:

- rule P, i.e., the rejection probability p of a new solution is proportional to the product of the differences between the current and the new solution in all their performance metrics;
- and rule W, i.e., p is proportional to the smallest value among the differences between the current and the new solution in all their performance metrics.

6.3.2 Evolutionary Algorithms

This section describes algorithms belonging to the second class defined in Sect. 6.2 in detail. The majority of these approaches is composed by variations of traditional genetic algorithms. The basic way a GA works is by improving an initial set of solutions (often randomly chosen), called population, by computing new solutions as combinations of existing solutions X picked with a probability p proportional to their fitness $f(X)$ [33].

6.3.2.1 Multiple Objective Genetic Local Search (MOGLS)

MOGLS [12] combines a typical class 2 methodology with one from class 1, that is, genetic algorithms with a local search. Those algorithms that combine multiple search methodologies are often referred to as "hybrid approaches." Each fundamental iteration step of MOGLS is composed of two sub-steps:

- first, new solutions are generated using genetic operations;
- and then, a local search is performed in the neighborhoods of these new solutions.

6.3.2.2 Ishibuchi-Murata Multi-objective Genetic Local Search (IMMOGLS)

IMMOGLS [13] is another hybrid algorithm that combines a genetic algorithm and a local search, just like MOGLS. In a multi-objective minimization problem, a solution is said to be non-dominated with respect to a set of solutions, if none of the other solutions score lower in all of the metrics one wants to minimize (or maximize). The iteration step of IMMOGLS, as for MOGLS, consists of both genetic operations and a local search but with three peculiar aspects:

- the fitness function used by the GA is a linear combination of the optimization metrics and the weights are chosen randomly at each iteration;
- the local search is limited to a certain (random) number of neighbors k;
- at each iteration, the current population is purged of any dominated solutions (this is referred to as an "elitist strategy").

6.3.2.3 Non-dominated Sorting Genetic Algorithm (NSGA)

NSGA [34] is a very successful application of the genetic approach to the problem of multi-objective optimization. The main insight regarding NSGA is the way in which the fitness of solutions is computed. All the individual in the current population are assigned fitness values on the basis of non-domination. All non-dominated solutions are assigned the same fitness value.

6.3.2.4 Controlled Non-dominated Sorting Genetic Algorithm (NSGA-II)

In [6], NSGA-II – an evolution of NSGA – is introduced. NSGA-II has two main peculiar aspects:

- NSGA-II is an elite-preserving algorithm; this means that non-dominated solutions cannot ever be removed from the current population;
- the sorting of solutions by non-domination also reduces computational complexity.

Genetic algorithms rely on several different genetic operators (mutation, crossover, etc.) to create new solutions as shown in Fig. 6.2. Thanks to GAs robustness, even the random selection of these operators usually leads to quality performance [24].

6.3.2.5 Pareto Memetic Algorithm (PMA)

PMA [15] is another member of the family of hybrid algorithms that combines GAs local searches. PMA differs from MOGLS and IMMOGLS in the way it selects the solutions for the crossover genetic operation. These are not drawn from the current population but from another set of size T that is the results of sampling

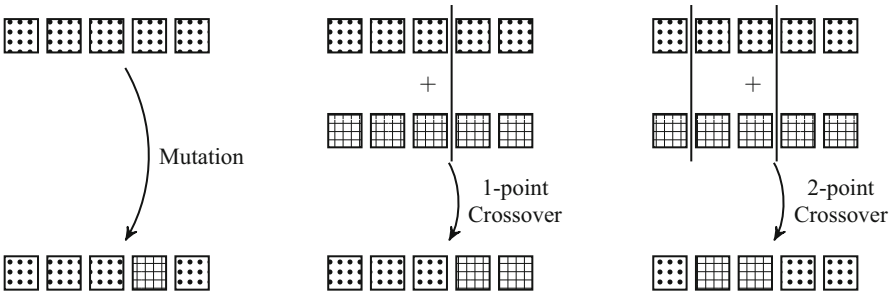


Fig. 6.2 Three of the most commonly used genetic operators: mutation, one-point, and two-point crossover

(with repetition) from the current population. The solutions that are chosen for recombination are the two with the best fitness in this new set.

6.3.2.6 Strength Pareto Evolutionary Algorithm (SPEA)

SPEA [38] is part of the larger family of heuristic search methods called evolutionary algorithms that also includes GAs. The three characteristic traits of SPEA are:

- all the non-dominated solutions are also stored in an auxiliary set/population;
- the fitness of a solution in the current population is determined only by comparison with the solutions in this auxiliary set [38];
- clustering is used to limit the size of the auxiliary population.

6.3.3 Statistical Approaches Without Domain Knowledge

This section describes algorithms belonging to the third class defined in Sect. 6.2 in detail.

6.3.3.1 Response Surface Pareto Iterative Refinement (ReSPIR)

ReSPIR [27] is a DSE tool that exploits two statistical/learning methodologies to infer the relationships between the tunable parameters of a system and its performance metrics. The benefit of this approach is that the number of evaluations needed to optimize a design is greatly reduced. These two pillars of ReSPIR are:

- DoE, a methodology that maximizes the information gained from a set of empirical trials;
- and RSMs, analytical representations of a performance metric reconstructed from the data. Several models can be applied for this problem: linear regression, Shepard-based interpolation, Artificial Neural Networks (ANNs), etc.

The main iteration step of ReSPIR consists of using DoE to define a set of experiments to perform, training the RSMs with the information collected, and finally producing an intermediate Pareto set.

6.3.4 Statistical Approaches with Domain Knowledge

This section describes algorithms belonging to the fourth class defined in Sect. 6.2 in detail.

6.3.4.1 Markov Decision Process Optimization (MDP)

The approach proposed by [3] is based on a framework for sequential decision making called Markov decision process. The components of an MDP are states, actions, stochastic transitions from state to state, and rewards (Fig. 6.3). The solution

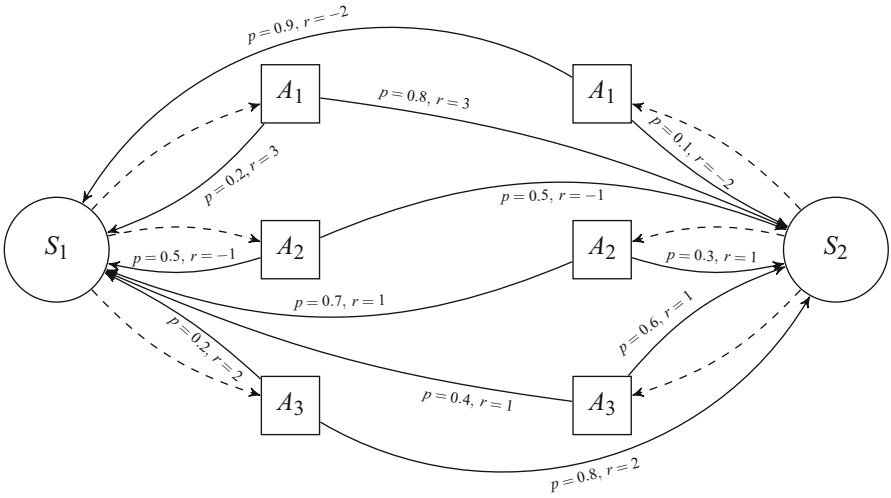


Fig. 6.3 An MDP with two states and three actions. Each arc going from an action A_x to a state S_y is associated with a probability p and a reward r

of an MDP is a strategy, i.e., the correct actions to perform in each one of the states to collect the largest amount of rewards [30] in a certain time horizon. In the formulation of [3], states are points in the design space (with their relative performance metrics), actions are changes in the tunable parameters, and rewards are improvements in the performance metrics. Stochastic transitions are initialized with uniform distribution and their likelihood are then refined throughout the execution of the algorithm. The main drawback of this approach is that it requires knowledge of the upper and lower bounds of each performance metric as a function of the tuning parameters. For this reason, MDP has a long setup time and cannot be used without thorough knowledge of the system to optimize.

6.3.4.2 Multi-objective Markov Decision Process (MOMDP)

MOMDP [1] is an improved version of MDP. The main difference lies in MOMDP novel exploration strategy. MDP, in fact, aggregated multiple performance metrics with a scalarizing function. By changing the value taken from a parameter called α (representing the weight(s) associated to each metric), it was possible to discover a Pareto curve for multiple separate objectives.

MOMDP, instead, uses a different approach: it maximizes (or minimizes) one of the objectives to derive a starting point and then builds the Pareto curve using a value function that selects a point that is close to the starting point, but improving it in at least one of the objectives. The process is repeated using the newly found point until a full Pareto front is discovered.

MOMDP also introduces a special action, called the leap of faith, that allows to avoid local minima by searching in the direction of high rewards, however unlikely. This action is performed when all actions fail to improve any of the metrics.

6.4 Experimental Comparison

Most class 1 and class 2 algorithm implementations can be found in the Multiple Objective MetaHeuristics Library in C++ (MOMHLib++) [16]. Multicube Explorer (M3Explorer) [37] also implements standard and enhanced versions of several well-known multi-objective optimization algorithms. Multicube Explorer provides some of the DSE algorithms in classes 1 and 3. Concerning the algorithms in class 4, MDP and MOMDP, they are evaluated using the original source code.

6.4.1 Objectives

This experimental comparison has multiple aims. Its three most important aspects are:

- determine the effort required to configure each algorithm for a given design space and how the characteristics of the design space influence the choice of the most effective exploration algorithm;
- determine the number of evaluations required by each algorithm to obtain an approximate Pareto set that meets certain quality requirements;
- and, finally, quantify the quality of the resulting Pareto set found by each algorithm.

A qualitative comparison of the 15 algorithms is presented in Table 6.3. Each algorithm was tested in the context of the same design space: a symmetric multi-processor platform running three different applications, shown in Fig. 6.4.

6.4.2 Benchmark Applications

Three applications (listed in Table 6.1) were used for testing, more specifically, two large applications and a small benchmark, for which exhaustive search was possible. *ffmpeg*, a video transcoder, was used to convert a small clip from MPEG-1 to MPEG-4, and *pigz*, a parallel compression algorithm, was used to compress a text file. The small benchmark consists of an implementation of Bailey's six-step FFT algorithm (*fft6*). All applications are data-parallel and are targeted toward a homogeneous shared-memory multi-processor platform (N processors accessing a common memory via bus). *ffmpeg* and *pigz* are implemented using pthreads; they create a set of working threads equal to the number of available processors and dispatch independent data to each thread. *fft6* uses OpenMP, with loop parallelization and static scheduling. These applications were chosen in order to display the maximum variability in their behavior, as they use a different synchronization mechanisms and require very different evaluation times. For situations in which it is especially important to identify the optimal mapping of complex, multi-application workloads,

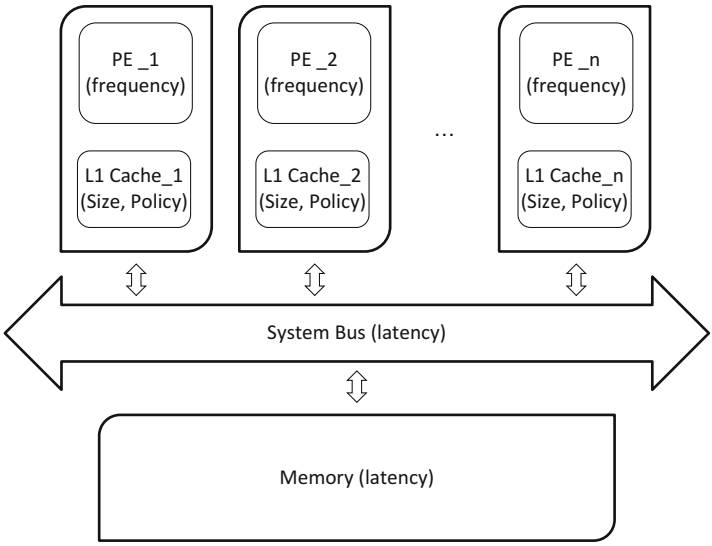


Fig. 6.4 The simulated multi-core processor architecture and its parameters (From [29])

Table 6.1 The three benchmark applications chosen to represent a significant spectrum of workloads

Application	Version	Source	Model	Synch.	Sim. Time	Description
pigz	2.1	C	pthreads	Condition	∼2 m	A parallel implementation of gzip
fft6	2.0	C/Fortran77	OpenMP	Barrier	∼30 s	Implementation of Bailey's 6-step fast Fourier transformation algorithm
ffmpeg	49.0.2	C	pthreads	semaphore	∼30 m	A fast video and audio converter

▸ Chap. 9, "Scenario-Based Design Space Exploration" explores thoroughly the domain of scenario-based DSE.

6.4.3 Target Computing Platform

The target platform consists of a collection of ARM9 cores with private caches and a shared memory (a comprehensive review of memory architectures and organizations can be found in ▸ Chap. 13, "Memory Architectures") interconnected by a simple system-bus model. Cache coherence is directory-based and implements the MESI protocol. The number of processing elements varies between one and eight. In the context of heterogeneous (e.g., big.LITTLE) multi-processor architectures, ▸ Chap. 8, "Architecture and Cross-Layer Design Space Exploration"

Table 6.2 The platform design space simulated using ReSP

Parameter name	Domain
# of PEs	{1,2,3,4,8}
PE frequency	{100,200,250,300,400,500} MHz
L1 cache size	{1,2,4,8,16,32} KByte(s)
Bus latency	{10,20,50,100} ns
Memory latency	{10,20,50,100} ns
L1 cache policy	{LRU, LRR, RANDOM}

examines how cross-layer optimization and predictive models can further enhance the DSE process.

The ReSP [2] open-source simulation environment was used to perform the simulations, providing a set of configurable parameters, listed in Table 6.2. ReSP provides values for execution time and power consumption, which were used as the performance metrics for all optimization algorithms in our experiments. For more on energy optimization, power, and thermal constraints in DSE, please refer to the methodologies in ▸ Chap. 10, "Design Space Exploration and Run-Time Adaptation for Multicore Resource Management Under Performance and Power Constraints".

The platform was explored using the parameters listed in Table 6.2 with a resulting design space of 8640 points (It is worth noting that bus and memory latency are not realistic parameters, but they enlarge the design space to better test the proposed algorithm. The linear dependence with performance prevents any strong biasing of the results.), comparable with similar works (e.g., 6144 points in [32]) and such that the exhaustive exploration of any medium/large application would require an unfeasibly long simulation time (e.g., roughly two months for *ffmpeg*). Even the full exploration of the simple *fft6* benchmark required six days of uninterrupted simulation. To gather sufficient data for a statistical analysis, each of the three benchmark applications was optimized ten times with each exploration algorithm ($N = 30$ executions for each algorithm).

6.4.4 Metrics to Evaluate Approximate Pareto Sets

According to [35], the quality of the result of a multi-objective optimization algorithm is twofold: (1) solutions should be as close as possible to the actual Pareto set and (2) solutions should be as diverse as possible. Therefore, no single metric is sufficient in assessing the quality of the discovered Pareto set.

Because of this reason, three metrics presented in [8] are used to compare the relative quality of the approximate Pareto sets obtained by the 15 algorithms under evaluation:

6.4.4.1 Average Distance from Reference Set

The Average Distance from Reference Set (ADRS) is used to compare the approximated Pareto sets with the best Pareto set (found combining the results of all

experiments). ADRS approximates the distance of the set under scrutiny from the Pareto-optimal front and should be minimized: an algorithm with low ADRS is very likely to have found a Pareto set that resembles the actual one.

As defined in [37], the ADRS between an approximate Pareto set Λ and a reference Pareto set Π is computed as:

$$\text{ADRS}(\Pi, \Lambda) = \frac{1}{|\Pi|} \sum_{-a \in \Pi} \left(\min_{-b \in \Lambda} \{\delta(-b, -a)\} \right) \tag{6.1}$$

where the δ function stands for:

$$\delta(-b, -a) = \max_{j=1,..,m} \left\{ 0, \frac{\phi_j(-a) - \phi_j(-b)}{\phi_j(-b)} \right\} \tag{6.2}$$

Parameter m is the number of objectives and $\phi_i(-a)$ is the value of the i-th objective metric measured in point $-a$.

6.4.4.2 Non-uniformity

Non-uniformity measures how solutions are distributed in the design space. A good search algorithm should look at all the different regions of a design space with equal attention. Lower non-uniformity means a more evenly distributed approximate Pareto set that better estimates the optimal Pareto set.

Given a normalized Pareto set $\bar{\Lambda}$, where d_i is defined as the Euclidean distance between two consecutive points ($i = 1, .., |\bar{\Lambda}| - 1$), and $\hat{d}$ is the average value of all the d's, non-uniformity [8] can be computed as:

$$\sum_{i=1}^{|\bar{\Lambda}|-1} \frac{|d_i - \hat{d}|}{\sqrt{m}(|\bar{\Lambda}| - 1)} \tag{6.3}$$

6.4.4.3 Concentration

Concentration measures the span of each Pareto set with respect to the range of the objectives. The lower the concentration, the higher the spread of the Pareto set and the better coverage of the range of objectives. It is important to observe that non-uniformity captures the behavior of a search algorithm in the design space, while concentration looks at points in the space of objective metrics.

Given a normalized Pareto set $\bar{\Lambda}$, where $\phi_i^{\min}$ is defined as $\min\{\phi_i(-a)\ s.t.\ -a \in \bar{\Lambda}\}$ and $\phi_i^{\max}$ is defined as $\max\{\phi_i(-a)\ s.t.\ -a \in \bar{\Lambda}\}$, concentration [8] can be computed as:

$$\prod_{i=1}^{m} \frac{1}{|\phi_i^{\max} - \phi_i^{\min}|} \tag{6.4}$$

6.4.5 Performance of the Algorithms Under Test

Having established the experimental context – that is, a set of benchmark applications and the target computing platform – and the performance metrics necessary to evaluate approximate Pareto sets, this section details the obtained results.

6.4.5.1 Initial Setup Effort and Parameter Sensitivity

All the examined algorithms differ in the way they converge to an approximate Pareto front, and the quality of their results depends on a number of different parameters, making a fair evaluation difficult to perform. There is no common rule for the choice of each algorithm's parameters: these range from 4 to 12, and they can be anything from integers to the choice of an interpolation function. The selection of the parameters that are optimal for a specific optimization problem requires either human expertise or the meta-exploration (also known as parameter screening) of the algorithm parameters space, i.e., running multiple explorations while changing the parameters to optimize the result. Either way, finding the optimal parameters usually requires trial and error. The "Effort" column of Table 6.3 qualitatively presents the tuning cost required by each algorithm.

Algorithms of classes 1 and 2 only require few parameters (such as population size, mutation factors, initial temperature, etc.), and they are generally robust to parameter choice. This means that small changes will not dramatically affect the outcomes of the exploration, although there is no guarantee that a given parameter choice will lead to optimal results. Their parameters (e.g., initial temperature for MOSA and NSGA-II) have no direct link to any characteristic of the design space and can be determined only by experience or guesswork, rendering the best

Table 6.3 A qualitative analysis of the chosen algorithms: setup effort, number of evaluations for 1% ADRS, number of Pareto points found, and scalability

Acronym	Class	Setup effort	Evaluations	Pareto points	Scalability
APRS [37]	1	⋆	⋆ ⋆ ⋆ ⋆ ⋆	⋆⋆	⋆
MOMSLS [16]	1	⋆	⋆ ⋆ ⋆	⋆	⋆ ⋆ ⋆
MOPSO [26]	1	⋆⋆	⋆ ⋆ ⋆⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆⋆
MOSA [36]	1	⋆ ⋆ ⋆	⋆ ⋆ ⋆⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆⋆
PSA [5]	1	⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆	⋆ ⋆ ⋆⋆
SMOSA [31]	1	⋆ ⋆ ⋆	⋆	⋆	⋆ ⋆ ⋆⋆
MOGLS [12]	2	⋆⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆⋆
IMMOGLS [13]	2	⋆⋆	⋆ ⋆ ⋆	⋆⋆	⋆ ⋆ ⋆⋆
NSGA [34]	2	⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆	⋆ ⋆ ⋆⋆
NSGA-II [6]	2	⋆ ⋆ ⋆	⋆ ⋆ ⋆⋆	⋆	⋆ ⋆ ⋆⋆
PMA [15]	2	⋆⋆	⋆ ⋆ ⋆	⋆⋆	⋆ ⋆ ⋆⋆
SPEA [38]	2	⋆⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆⋆
ReSPIR [27]	3	⋆ ⋆ ⋆⋆	⋆⋆	⋆ ⋆ ⋆⋆	⋆ ⋆ ⋆
MDP [3]	4	⋆ ⋆ ⋆ ⋆ ⋆	⋆	⋆	⋆⋆
MOMDP [1]	4	⋆ ⋆ ⋆ ⋆ ⋆	⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆

combination very difficult to obtain without screening and additional evaluations. For the comparison in this chapter, the best parameters were determined via screening, which required running several thousand evaluations.

Algorithms like APRS have a minimal setup effort as they do not require any special tuning and rely on pre-determined heuristics. It is worth noting that both class 1 and 2 algorithms do not guarantee convergence to the optimal Pareto set and require the user to specify a maximum number of iterations in addition to any other stopping condition (e.g., when the results do not vary for more than two iterations).

Algorithms in class 3 demand a greater setup effort: the choice of proper metamodels for the design space requires some expertise and an initial screening (i.e., additional evaluations) to properly determine which parameters are the most significant. Each metamodel needs specific additional parameters that are loosely linked to the designer's expertise of the design space. For the comparison in this chapter, the central composite design using a Neural Network (NN) interpolator was used, as suggested by the results in [27]. However, the NN produced results with a very high variance, with ADRS ranging from 0 to 160%, making the use of this interpolator impractical. The Shepard interpolation was found to be much more effective, although it required to determine the value of a *power* parameter, which expresses how jagged is the response surface of the design space. A low value of this *power* parameter will produce a smooth interpolation, while a higher value could better follow a more jagged curve, but could also introduce overfitting. The results of [27] were effectively replicated in the considered design space using a power of 16, which was found by parameter screening ($\sim 10^3$ evaluations).

Finally, algorithms in class 4 require to set bounds to the effects of parameter variations on the configuration's metrics. The quantification of these bounds is left to the designer's experience or can be determined via statistical modeling. This means detailed analysis of each design space and a large setup effort. The main difference between MDP and MOMDP is that the former is fundamentally a single-objective optimization algorithm. To effectively discover a Pareto set, MDP "sweeps" the design space according to a set of scalarizing values that express the desired trade-off between the different objective functions. To determine the size and values of these scalarizing values for the design space under evaluation, hundreds of additional evaluations are required. MOMDP does not require this screening phase, and its only parameter (the accuracy λ) is in fact the average simulation error and can be chosen without effort.

It is worth noting that designer experience can reduce or remove the need for parameter discovery activities for all the aforementioned algorithms.

6.4.5.2 Convergence Rate

For the comparison in this chapter, each algorithm's parameters were optimally tuned but the evaluations needed for screening and initial parameter estimation are **not** accounted for. Concerning ADRS, since exhaustive search is not possible, the reference Pareto set is the best Pareto set generated compounding all evaluations performed by all algorithms, which covered a sizable portion of the entire design space (around 30%).

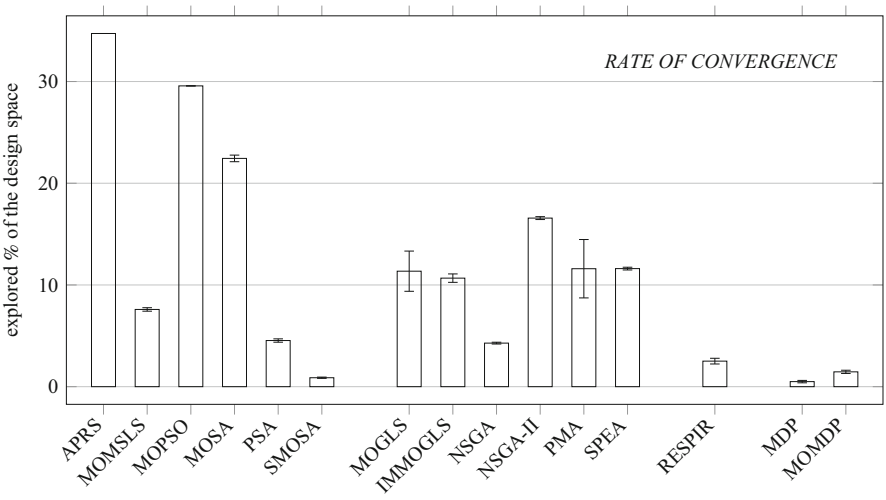


Fig. 6.5 The percentage of points in the design space evaluated by each algorithm for similar levels of accuracy, that is, ∼1% (From [29])

Figure 6.5 shows the percentage of the design space (i.e., the number of evaluations divided by the number of points in the design space) explored by each algorithm in order to reach an ADRS of approximately 1% on average. Note that it was not possible for all the algorithms to converge to the exact same quality result, and some algorithms show high variability. In fact, SMOSA and NSGA often do not converge to acceptable solutions. The final accuracy values obtained are shown in Fig. 6.6. Please note that the histograms and the error bars in Figs. 6.5, 6.6, 6.7, 6.8 and 6.9 show average values and standard deviations, respectively, for each algorithm over 30 experiments. No negative percentage actually resulted during the experiments.

Figure 6.5 shows that the improvement can be worth the extra setup effort for class 3 and 4 algorithms: MDP, MOMDP, and RESPIR have a factor 10 reduction in the number of evaluations and a much tighter convergence (i.e., smaller variance of the results). Concerning class 2 algorithms, the performance is very similar, with IMMOGLS appearing to have the best combination of accuracy, number of evaluations, and variance. APRS still provides excellent results given the zero-effort setup, although with at least twice as many evaluations when compared to class 2 algorithms.

6.4.5.3 Quality of the Approximate Pareto Set

Figure 6.7 shows the number of Pareto points found by each algorithm, normalized by the average number found (for each benchmark).

The performance of most algorithms does not appear to show any statistically significant difference, with the exception of RESPIR, which finds, on average, 25% more points than all the other algorithms but with a slightly higher variance. Once

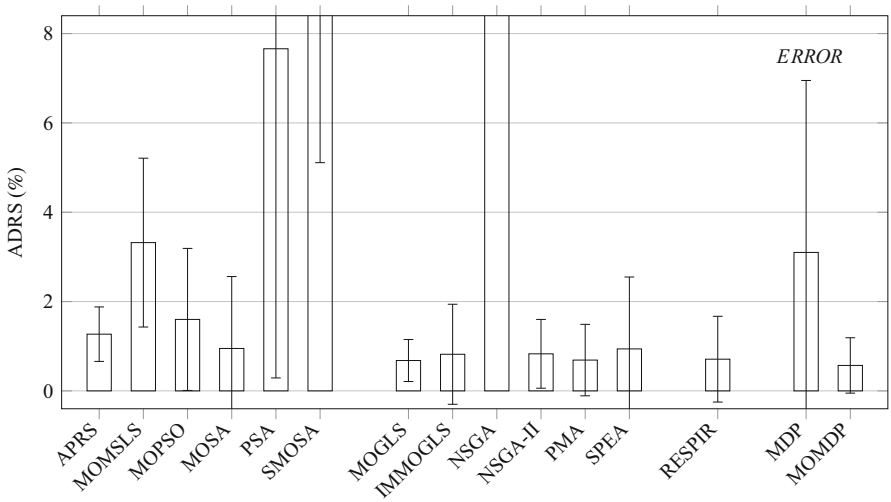


Fig. 6.6 Accuracy (ADRS) reached by each algorithm at convergence (From [29])

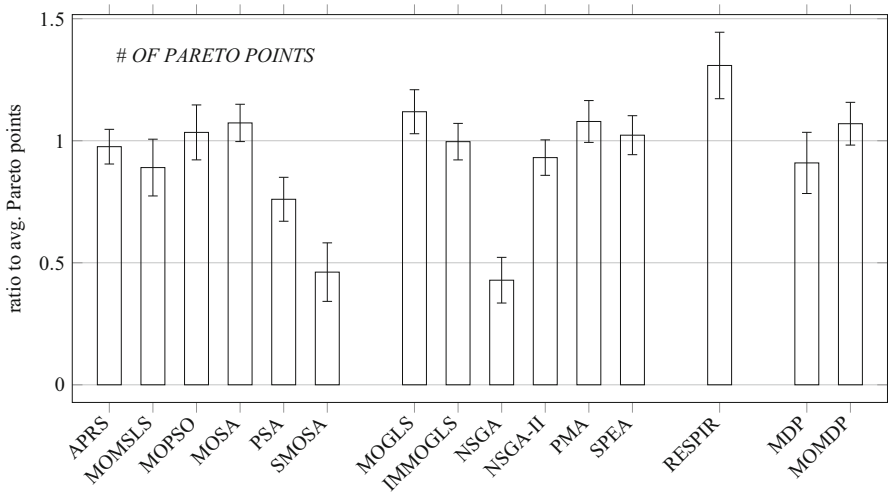


Fig. 6.7 The average number of points in the approximate Pareto set found by each algorithm, normalized with the average for each benchmark (From [29])

again, SMOSA and NSGA display the poorest results. It is worth noting that some of the points found by RESPIR are Pareto-covered by the points found by the other algorithms: the number of points on the actual Pareto curve is smaller than what was found by RESPIR.

Concerning non-uniformity and concentration, all algorithms behave similarly, satisfyingly covering the design space and without concentrating on specific

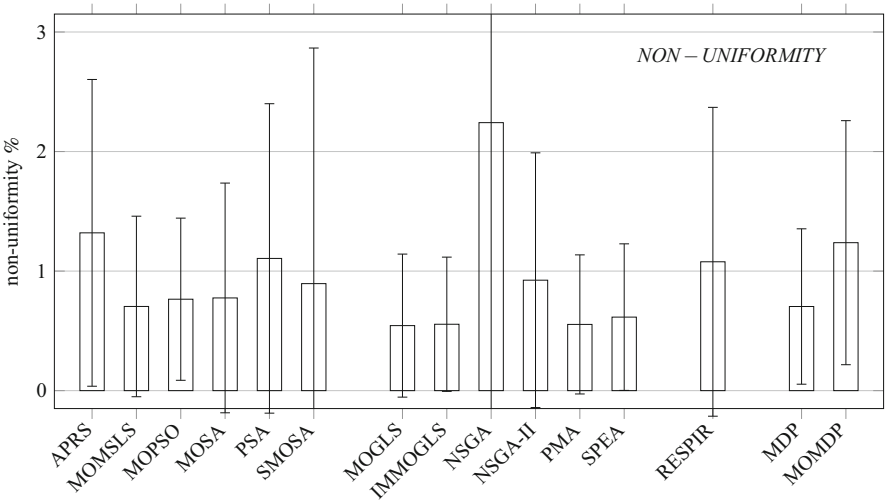


Fig. 6.8 The non-uniformity of the distribution of the points found in the approximate Pareto set found by each algorithm (From [29])

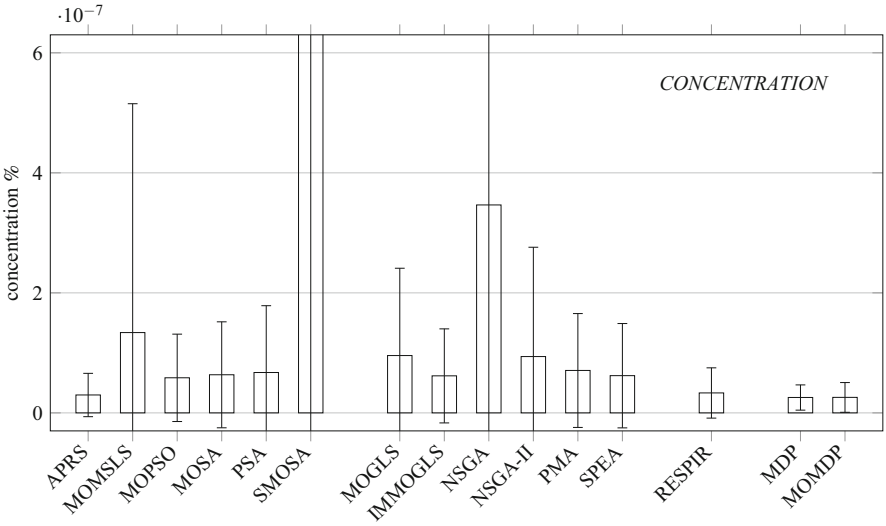


Fig. 6.9 The concentration of the points found in the approximate Pareto set found by each algorithm (From [29])

areas. Again, no statistically significant difference between the algorithms can be observed, with the only exception of SMOSA and NSGA, which produced the worst results as well as higher variance. Results are presented in Figs. 6.8, 6.9 and 6.10.

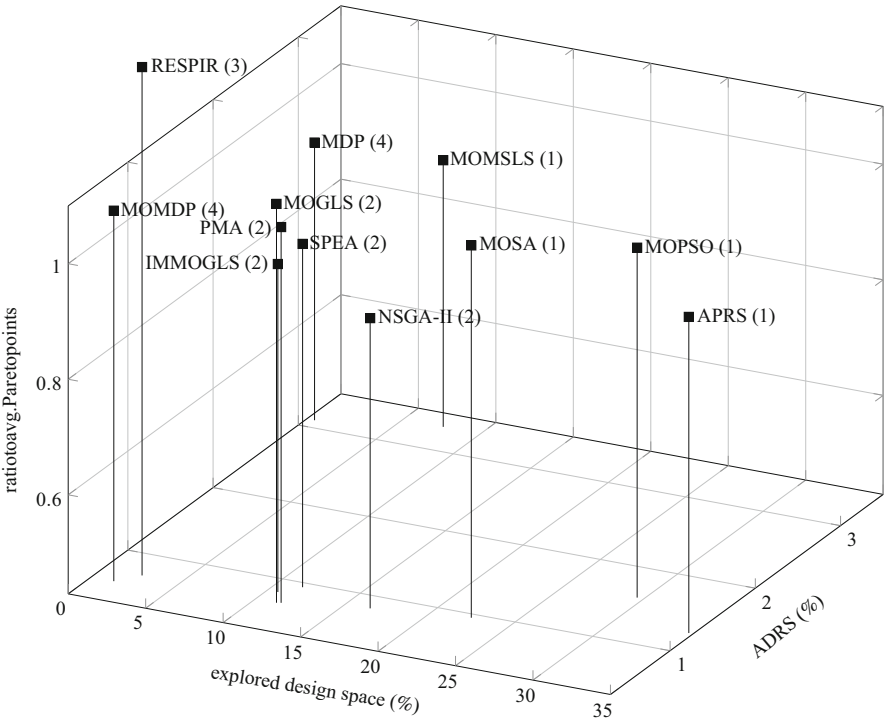


Fig. 6.10 3D representation of the algorithms performance showing % of the design space explored in order to reach convergence, the number of Pareto points (w.r.t the average number), and the ADRS error metric (NSGA, PSA, and SMOSA are omitted because of their large ADRS). The number after each algorithm name indicates the its class (from [29])

6.4.5.4 Scalability

The scalability column of Table 6.3 tells how an algorithm's performance degrades with the increasing size of the design space: more scalable algorithms can be applied to more complex design spaces with lower effort. To test the effective scalability of each algorithm, the size of the design space was progressively increased, starting with three parameters and adding the remaining three one by one.

Most algorithms of classes 1 and 2 are very scalable: the number of parameters – or the number of values per parameter – does not affect the overall result, even though the number of additional evaluations needed grows proportionally with the number of parameters (i.e., remains around ∼10–15% of the design space).

Although APRS require little setup effort, its applicability to large design spaces cannot be guaranteed. In fact, the already very high number of evaluations required increases exponentially with the design space size, practically limiting its use to small design spaces with fast evaluations.

RESPIR (class 3) scales well to design spaces with many parameters, but only if few values per parameter are present. This is due to one of the main limitations

of central composite design: it can only consider three levels for each parameter, therefore reducing the accuracy of the method in presence of many parameter values, especially if they lead to non-linear behavior. The number of evaluations for convergence remains ∼4% of the design space.

Finally, class 4 algorithms scale orthogonally with respect to class 3: they scale well with the number of values per parameter, but not when the number of parameters increases. While adding a parameter requires defining new bounds, adding new values comes without effort, and the number of evaluations needed increases less than linearly [3]. One drawback of MDP when compared to MOMDP is that it requires the estimation of an additional parameter (α; see [3]) when increasing the size of the design space, which might require additional evaluations.

6.5 Discussion

Selection of the best algorithm for a particular application is not an easy task, and it requires trading-off setup effort, scalability, expected number of simulations, and accuracy. Looking at the result from Sect. 6.4, one can draw some general guidelines according to the cost of each evaluation (in terms, e.g., of simulation time) and the size of the design space.

High evaluation costs make algorithms requiring a low number of simulations more appealing, even if they have a high upfront setup cost. On the contrary, if evaluations have moderate costs, one might want to trade-off a higher number of evaluations for a no-effort setup. Similarly, large design spaces favor scalable algorithms, while smaller spaces do not justify the extra work required to apply sophisticated algorithms.

In order to estimate the *actual* convergence time of an algorithm i, one must take into consideration the design space size ($|S|$ points), the time required by each simulation/evaluation (T_{sim} seconds), the percentage of the design space explored before reaching convergence (ν_i), and the setup time T_i^{setup} of the algorithm:

$$T_i^{\text{actual}} = (\nu_i \times |S| \times T_{\text{sim}}) + T_i^{\text{setup}} \tag{6.5}$$

The values of ν_i's are reported in Fig. 6.5. Regarding the T_i^{setup} values, the qualitative information in Table 6.3 was translated into a 10′- to 30-h range: observations showed that algorithms having one "effort star" can be set up in a few minutes, while algorithms with five "effort stars" require more than a day of work.

Figure 6.11 presents four plots: the size of the design space appears on the x axis while the time required by each simulation appears on the y axis. In each plot, areas are labeled with the name of the most suitable algorithm according to Eq. 6.5. Subplots (a) and (b) report the result for algorithms able to obtain ADRS of about 1%, whether domain knowledge is available (a) or not (b). Subplots (c) and (d) relax the ADRS requirement to about 5%.

Whether or not domain knowledge is available, multi-objective multiple start local search (MOMSLS) and adaptive windows Pareto random search (APRS) are

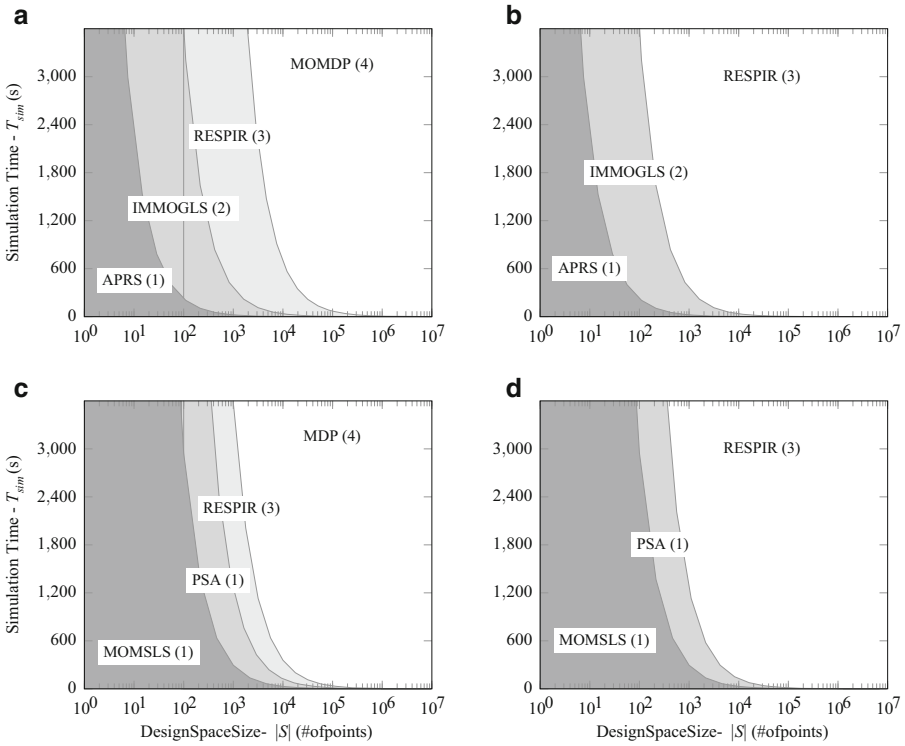


Fig. 6.11 Recommended algorithms for different design space size, simulation time and desired ADRS, whether domain knowledge is available or not. The number after each algorithm name indicates its class (From [29]). (**a**) ADRS $\sim$ 1%, w/ domain knowledge. (**b**) ADRS $\sim$ 1%, w/o domain knowledge. (**c**) ADRS $\sim$ 5%, w/ domain knowledge. (**d**) ADRS $\sim$ 5%, w/o domain knowledge

the most appealing solutions for small to medium design spaces with inexpensive evaluations. As explained in Sect. 6.4, APRS is a heuristic algorithm that requires very little setup effort, making it the ideal choice when simplicity is valued more than raw performance. MOMSLS is considerably faster but also more likely to produce a larger ADRS.

In large design spaces with high cost evaluations – if domain knowledge is available – the multi-objective MDP algorithm (MOMDP) [1] clearly grants better performance, both in terms of fast convergence to a very small ADRS and quality of the Pareto set.

When one cannot obtain or exploit domain knowledge, the choice is split between the very high-quality Response Surface Pareto Iterative Refinement (RESPIR) algorithm and the Ishibuchi-Murata MO Genetic Local Search (IMMOGLS) algorithm. Both these algorithms are suited for very large design spaces.

Table 6.4 Recommended algorithms for space size and evaluation effort

	Design space size w/ knowledge		
Eval cost	Small	Medium	Large
Low	APRS	PMA	IMMOGLS
Medium	PMA	RESPIR	RESPIR
High	RESPIR	MOMDP	MOMDP
	Design space size w/o knowledge		
Eval cost	Small	Medium	Large
Low	APRS	PMA	IMMOGLS
Medium	PMA	RESPIR	IMMOGLS
High	RESPIR	RESPIR	RESPIR

However, it is worth noting that the IMMOGLS algorithm usually requires a larger number of evaluations, therefore is not recommended when dealing with high cost simulation. Pareto Simulated Annealing (PSA) is a valid alternative to IMMOGLS when a larger ADRS is acceptable.

What one can conclude from Fig. 6.11 is that algorithms with small setup times (i.e., the ones in classes 1 and 2) are especially suitable for simple problems with relatively small design spaces and/or short simulation times. On the other hand, complex algorithms in classes 3 and 4 usually compensate for their longer configuration times when the exploration problem is sufficiently challenging.

These recommendations (summarized in Table 6.4) are qualitative, but do take into account all the parameters discussed in Sect. 6.4.

6.6 Existing Frameworks

All the algorithms presented in this chapter have an open-source implementation. There are many optimization tools that can be used or adapted for design space exploration. It is generally advisable to use an existing, well-tested implementation of one of these algorithms instead of going for the homebrew solution. The reason is that the most common open-source frameworks are used by many developers, and many of the issues and bugs have been found just by the sheer volume of users.

Most frameworks are available in the form of *libraries* to be used with a specific language or development environment. In the following, examples of existing frameworks are given, providing a short analysis of their strong and weak points.

6.6.1 jMetal

jMetal [7] is an object-oriented Java-based framework for multi-objective optimization with metaheuristics. It provides 12 different multi-objective algorithms, as well as some single-objective metaheuristics. jMetal is one of the most popular frameworks available and has a number of advantages:

- It is based on Java, that is, it is platform independent and the API is easily accessible by a programmer.
- It provides a graphical user interface and a set of test problems with quality indicators.
- It provides support for the parallel execution for a subset of its algorithms.

The main disadvantage of jMetal arises from one of its strong points: being written in Java, jMetal does not have optimal performance, and its parallel performance does not scale very well. Despite this, jMetal is one of the most comprehensive existing frameworks, and its utilization is recommended if the algorithm of interest is present in its library.

6.6.2 PaGMO/PyGMO

PaGMO (https://github.com/esa/pagmo/) is a C++ optimization framework initially developed by the European Space Agency for the optimization of interplanetary trajectories. The framework focuses on novel algorithms, parallelism, and performance. Compared to jMetal, PaGMO provides similar algorithms, but all of them are tuned for massively parallel execution. Therefore, PaGMO has an edge in terms of exploration performance. As jMetal, PaGMO comes with sample problems, metrics, and extension abilities. PaGMO also has Python bindings (PyGMO), which make it accessible with very simple scripts.

The main disadvantage of PaGMO is that despite having been in developing for a few years, it is not as user-friendly as jMetal: the installation is only from source, there is no graphical user interface, and its documentation incomplete. PaGMO is therefore recommended to the more experienced programmer or to the user with extreme need for performance.

6.6.3 MOHMLib++

The Multiple Objective MetaHeuristics Library in C++ (MOMHLib++) [16] is a C++ library providing 15 optimization algorithms. It is not as developed or maintained as jMetal or PaGMO, but it has few dependencies and it is very simple to use.

MOHMLib++ has no graphical user interface, no support for massively parallel execution, and no scripting language bindings. However, its small size and simplicity are usually very attractive to an unambitious developer looking to find a quick solution to an optimization or design exploration problem.

6.6.4 NASA

NASA (Non Ad-hoc Search Algorithm) [17] is not a collection of search algorithms but rather a DSE framework characterized by its modularity. Because of this reason, NASA can be used orthogonally and together with the strategies implemented, for

example, by MOHMLib++. The infrastructure of NASA is implemented in C++ and consists of several modules (including a search module, a feasibility checker, a simulator, and an evaluator) that can be extended or replaced by its user in plug-and-play fashion.

The main advantage of NASA is, indeed, its modularity. NASA allows the designer to fully decouple important functionalities (such as the choice of a search algorithm and the evaluation of multiple performance metrics) with the aid of only three *interface files*. Moreover, NASA is already integrated with two widely used system-level simulators – CASSE and Sesame – and it supports the parallel exploration of design space dimensions that are deemed independent by the designer. A shortcoming of NASA is the lack of pre-implemented search methodologies other than a proprietary GA, making the joint use of one of the previous frameworks a must.

6.7 Conclusions

Concluding, this chapter presented a classification and comparative analysis of 15 of the best and most recent multi-objective design exploration algorithms. The algorithms were applied to the exploration of a multi-processor platform and they were compared for setup effort, number of evaluations, quality of the resulting approximate Pareto set, and scalability. The obtained results were then used as guidelines for the choice of the algorithm best suited to the properties of a target design space. In particular, the experiments determined the most promising algorithms when considering design space size and evaluation effort. Finally, a list of reusable, open frameworks implementing DSE optimization strategies is provided.

References

1. Beltrame G, Nicolescu G (2011, in press) A multi-objective decision-theoretic exploration algorithm for platform-based design. In: Proceedings of design, automation and test in Europe (DATE)
2. Beltrame G, Fossati L, Sciuto D (2009) ReSP: a nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration. IEEE Trans Comput Aided Des Integr Circuits Syst 28(12):1857–1869
3. Beltrame G, Fossati L, Sciuto D (2010) Decision-theoretic design space exploration of multiprocessor platforms. IEEE Trans Comput Aided Des Integr Circuits Syst 29(7):1083–1095. doi: 10.1109/TCAD.2010.2049053
4. Coello CA (2000) An updated survey of ga-based multiobjective optimization techniques. ACM Comput Surv 32(2):109–143. doi: 10.1145/358923.358929
5. Czyżżak P, Jaszkiewicz A (1998) Pareto simulated annealing–a metaheuristic technique for multiple-objective combinatorial optimization. J Multi-Criteria Decis Anal 7(1):34–47. doi: 10.1002/(SICI)1099-1360(199801)7:1<34::AID-MCDA161>3.0.CO;2-6
6. Deb K, Goel T (2001) Controlled elitist non-dominated sorting genetic algorithms for better convergence. In: Zitzler E, Thiele L, Deb K, Coello Coello C, Corne D (eds) Evolutionary multi-criterion optimization. Lecture notes in Computer Science, vol 1993. Springer, Heidelberg, pp 67–81. doi: 10.1007/3-540-44719-9_5

7. Durillo JJ, Nebro AJ (2011) jMetal: a java framework for multi-objective optimization. Adv Eng Softw 42:760–771
8. Erbas C (2006) System-level modelling and design space exploration for multiprocessor embedded system-on-chip architectures. Amsterdam University Press, Amsterdam
9. Fonseca CM, Fleming PJ (1995) An overview of evolutionary algorithms in multiobjective optimization. Evol Comput 3(1):1–16. doi: 10.1162/evco.1995.3.1.1
10. Fornaciari W, Sciuto D, Silvano C, Zaccaria V (2002) A sensitivity-based design space exploration methodology for embedded systems. Des Autom Embed Syst 7(1):7–33. doi: 10.1023/A:1019791213967
11. Givargis T, Vahid F, Henkel J (2001) System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In: 2001 IEEE/ACM international conference on computer aided design, ICCAD 2001, pp 25–30. doi: 10.1109/ICCAD.2001.968593
12. Ishibuchi H, Murata T (1996) Multi-objective genetic local search algorithm. In: Proceedings of IEEE international conference on evolutionary computation, pp 119–124. doi: 10.1109/ICEC.1996.542345
13. Ishibuchi H, Murata T (1998) A multi-objective genetic local search algorithm and its application to flowshop scheduling. IEEE Trans Syst Man Cybern C Appl Rev 28(3):392–403. doi: 10.1109/5326.704576
14. Jaddoe S, Pimentel AD (2008) Signature-based calibration of analytical system-level performance models. In: Proceedings of the 8th international workshop on embedded computer systems: architectures, modeling, and simulation SAMOS'08. Springer, Heidelberg, pp 268–278
15. Jaszkiewicz A (2004) A comparative study of multiple-objective metaheuristics on the bi-objective set covering problem and the pareto memetic algorithm. Ann Oper Res 131:135–158. doi: 10.1023/B:ANOR.0000039516.50069.5b
16. Jaszkiewicz A, Dabrowski G (2005) MOMH: multiple objective meta heuristics. Available at the web site http://home.gna.org/momh/
17. Jia ZJ, Bautista T, Núñez A, Pimentel AD, Thompson M (2013) A system-level infrastructure for multidimensional MP-SoC design space co-exploration. ACM Trans Embed Comput Syst 13(1s):27:1–27:26. doi: 10.1145/2536747.2536749
18. Kaelbling LP, Littman ML, Moore AP (1996) Reinforcement learning: a survey. J Artif Intell Res 4:237–285
19. Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings IEEE international conference on neural networks, vol 4, pp 1942–1948. doi: 10.1109/ICNN.1995.488968
20. Lukasiewycz M, Glay M, Haubelt C, Teich J (2008) Efficient symbolic multi-objective design space exploration. In: ASP-DAC '08: proceedings of the 2008 Asia and South Pacific design automation conference. IEEE Computer Society Press, Seoul, pp 691–696
21. Mariani G, Brankovic A, Palermo G, Jovic J, Zaccaria V, Silvano C (2010) A correlation-based design space exploration methodology for multi-processor systems-on-chip. In: 2010 47th ACM/IEEE design automation conference (DAC), pp 120–125
22. Mohanty S, Prasanna VK, Neema S, Davis J (2002) Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. SIGPLAN Not 37(7):18–27
23. Nebro A, Durillo J, Coello C (2013) Analysis of leader selection strategies in a multi-objective particle swarm optimizer. In: 2013 IEEE congress on evolutionary computation (CEC), pp 3153–3160. doi: 10.1109/CEC.2013.6557955
24. Nebro AJ, Durillo JJ, Machín M, Coello Coello CA, Dorronsoro B (2013) A study of the combination of variation operators in the NSGA-II algorithm. In: Advances in artificial intelligence: 15th conference of the Spanish association for artificial intelligence, CAEPIA 2013, Madrid, 17–20 Sept 2013. Proceedings. Springer, Heidelberg, pp 269–278. doi: 10.1007/978-3-642-40643-0_28
25. Okabe T, Jin Y, Sendhoff B (2003) A critical survey of performance indices for multi-objective optimisation. In: The 2003 congress on, evolutionary computation, CEC '03, vol 2, pp 878–885. doi: 10.1109/CEC.2003.1299759

26. Palermo G, Silvano C, Zaccaria V (2008) Discrete particle swarm optimization for multi-objective design space exploration. In: 11th EUROMICRO conference on digital system design architectures, methods and tools, DSD'08, pp 641–644. doi: 10.1109/DSD.2008.21
27. Palermo G, Silvano C, Zaccaria V (2009) ReSPIR: a response surface-based pareto iterative refinement for application-specific design space exploration. IEEE Trans Comput Aided Des Integr Circuits Syst 28(12):1816–1829. doi: 10.1109/TCAD.2009.2028681
28. Palesi M, Givargis T (2002) Multi-objective design space exploration using genetic algorithms. In: CODES '02: Proceedings of the tenth international symposium on hardware/software codesign. ACM, Colorado, pp 67–72. doi: 10.1145/774789.774804
29. Panerati J, Beltrame G (2014) A comparative evaluation of multi-objective exploration algorithms for high-level design. ACM Trans Des Autom Electron Syst 19(2):15:1–15:22. doi: 10.1145/2566669
30. Russell SJ, Norvig P (1995) Artificial intelligence: a modern approach, 1st edn. Prentice Hall, Upper Saddle River
31. Serafini P (1994) Simulated annealing for multi objective optimization problems. In: Tzeng G, Wang H, Wen U, Yu P (eds) Multiple criteria decision making. Springer, New York, pp 283–292. doi: 10.1007/978-1-4612-2666-6_29
32. Sheldon D, Vahid F, Lonardi S (2007) Soft-core processor customization using the design of experiments paradigm. In: DATE conference, pp 1–6
33. Sivanandam SN, Deepa SN (2007) Introduction to genetic algorithms, 1st edn. Springer, Berlin/New York
34. Srinivas N, Deb K (1994) Muiltiobjective optimization using nondominated sorting in genetic algorithms. Evol Comput 2(3):221–248. doi: 10.1162/evco.1994.2.3.221
35. Taghavi T, Pimentel AD (2011) Design metrics and visualization techniques for analyzing the performance of moeas in DSE. In: ICSAMOS, pp 67–76
36. Ulungu E, Teghem J, Fortemps P, Tuyttens D (1999) MOSA method: a tool for solving multiobjective combinatorial optimization problems. J Multi-Criteria Decis Anal 8(4):221–236. doi: 10.1002/(SICI)1099-1360(199907)8:4<221::AID-MCDA247>3.0.CO;2-O
37. Zaccaria V, Palermo G, Castro F, Silvano C, Mariani G (2010) Multicube explorer: an open source framework for design space exploration of chip multi-processors. In: 2PARMA: proceedings of the workshop on parallel programming and run-time management techniques for many-core architectures, Hannover
38. Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. IEEE Trans Evol Comput 3(4):257–271. doi: 10.1109/4235.797969

Hybrid Optimization Techniques for System-Level Design Space Exploration

7

Michael Glaß, Jürgen Teich, Martin Lukasiewycz, and Felix Reimann

Abstract

Embedded system design requires to solve synthesis steps that consist of resource allocation, task binding, data routing, and scheduling. These synthesis steps typically occur several times throughout the entire design cycle and necessitate similar concepts even at different levels of abstraction. In order to cope with the large design space, fully automatic Design Space Exploration (DSE) techniques might be applied. In practice, the high complexity of these synthesis steps requires efficient approaches that also perform well in the presence of stringent design constraints. Those constraints may render vast areas in the search space infeasible with only a fraction of feasible implementations that are sparsely distributed. This is a serious problem for metaheuristics that are popular for DSE of electronic hardware/software systems, since they are faced with large areas of infeasible implementations where no gradual improvement is possible. In this chapter, we present an approach that combines metaheuristic optimization with search algorithms to solve the problem of Hardware/Software Codesign (HSCD) including allocation, binding, and scheduling. This hybrid optimization uses powerful search algorithms to determine feasible implementations This avoids an exploration of infeasible areas and, thus, enables a gradual improvement as

M. Glaß
Institute of Embedded Systems/Real-Time Systems at Ulm University, Ulm, Germany
e-mail: michael.glass@uni-ulm.de

J. Teich
Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany
e-mail: juergen.teich@fau.de

M. Lukasiewycz
Robert Bosch GmbH, Corporate Research, Renningen, Germany
e-mail: martin.lukasiewycz@de.bosch.com

F. Reimann
Audi Electronics Venture GmbH, Gaimersheim, Germany
e-mail: felix.reimann@audi.de

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_8

required for efficient metaheuristic optimization. Two methods are presented that can be applied to both, problems with linear as well as non-linear constraints, the latter being particularly intended to address aspects such as timeliness or reliability which cannot be approximated by linear constraints in a sound fashion. The chapter is concluded with several examples for a successful use of the introduced techniques in different application domains.

Acronyms

BIST	Built-In Self-Test
DPLL	Davis-Putnam-Logemann-Loveland
DSE	Design Space Exploration
EA	Evolutionary Algorithm
E/E	Electric and Electronic
ESL	Electronic System Level
HSCD	Hardware/Software Codesign
ILP	Integer Linear Program
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
PB	Pseudo-Boolean
SAT	Boolean Satisfiability
SMT	Satisfiability Modulo Theories

Contents

7.1 Introduction and Motivation

The design of electronic embedded systems typically requires to solve the crucial synthesis steps of resource allocation, task binding, data routing, and scheduling. Those basic steps can even re-occur throughout the design cycle [30] and necessitate similar concepts even at different levels of abstraction. Here, a major problem for

design space exploration is typically not just a vast design space, but the high complexity of these synthesis steps (NP-complete) that becomes even more severe in the presence of stringent design constraints. Those constraints may render many possible system implementations infeasible, such that vast areas in the search space are infeasible with feasible implementations populating the space only sparsely. This is a tremendous problem for metaheuristics that are popular for *Design Space Exploration* (DSE) (see ▸ Chap. 6, "Optimization Strategies in Design Space Exploration"), since they are faced with large areas of infeasible implementations where no gradual improvement is possible.

This chapter describes an approach to overcome this problem. The main idea is to employ a backtracking-based search algorithm, i.e., a *Pseudo-Boolean* (PB) solver, to obtain feasible implementations only. This search algorithm is controlled by a metaheuristic optimization technique, i.e., an *Evolutionary Algorithm* (EA) to (a) enable an efficient exploration even in large and sparse search spaces and (b) search for implementations that are optimized with respect to multiple non-functional design objectives such as timeliness, reliability, and/or power consumption. This combination is termed *SAT decoding* and can be considered a *hybrid optimization* approach which is exemplified for system-level DSE of electronic hardware/software systems in this chapter. However, the employed search algorithm is only capable of handling linear or linearizable design constraints in a Boolean domain.

In the presence of constraints that cannot be efficiently linearized and reduced to the Boolean domain – such as a maximum end-to-end delay of an application with tasks mapped to multiple resources – the approach will again deliver infeasible implementations if these non-linear constraints are ignored. A technique to overcome this drawback is as well presented in this chapter. The basic idea is to integrate analysis techniques for such non-functional constraints and incrementally determine linear constraints for the pseudo-Boolean solver. By this way, the solver is capable of learning which implementations are infeasible. The proposed hybrid Design Space Exploration (DSE) approach is not only applicable at the Electronic System Level (ESL), but may be applied also at other levels of hardware and software synthesis in embedded system design.

This chapter is structured into three main sections: Sect. 7.2 introduces required fundamentals as well as the mathematical formulation of the synthesis problem to be solved. Section 7.3 presents the hybrid optimization technique SAT decoding that can consider linear constraints. An extension of SAT decoding which considers non-linear constraints termed *SMT decoding* is discussed subsequently in Sect. 7.4. Examples of applications of the introduced techniques for further reading are presented in Sect. 7.5 before the chapter is concluded in Sect. 7.6.

7.2 Fundamentals and Problem Formulation

The first work on Hardware/Software Codesign (HSCD) can be found in [22] which considers the problem of concurrently defining a multi-processor system's topology, a binding of tasks to processors, and their scheduling. Since, the problem of allocating hardware and software components, followed by binding tasks to

either hardware or software, became known as *hardware/software codesign*. Many initial works consider the codesign problem a *bipartition problem*, i.e., a task is assigned either to a processor and executed as software or to one dedicated hardware accelerator. This notion is generalized to heterogeneous hardware/software architectures with multiple components in [29] under the term *system-level synthesis*. In [1], same authors prove that this system-level synthesis problem is an NP-complete problem. For an in-depth discussion of the historical roots of hardware/software codesign, see ▸ Chap. 1, "Introduction to Hardware/Software Codesign". For comprehensive overviews on system-level synthesis techniques, interested readers can refer to [6, 28].

In the following sections, we introduce a well-established model for the system-level synthesis problem which is very suitable for (networked) embedded systems. Afterward, we also discuss the problem from an optimization perspective where system synthesis can be considered a *constrained combinatorial optimization problem* and give a brief introduction of common optimization approaches and constraint-handling techniques.

7.2.1 System Model and the System-Level Synthesis Problem

In the following section, we introduce a formal graph-based system model and the respective system-level synthesis problem as a variation and extension of the original model from [1] that is proposed in [21].

7.2.1.1 System Model

A system model ς termed *specification* is given in a graph-based fashion that distinguishes between *application* (modeled as an *application graph* G_T) and an *architecture* (modeled as an *architecture graph* G_R). The relation between application and architecture – indicating each possible binding of a task of the application for execution on a resource of the architecture – is modeled by means of a set of *mapping edges* E_M.

- The application is given by a bipartite directed graph $G_T(T, E_T)$ with $T = P \cup C$. The vertices T are either process tasks $p \in P$ or communication tasks $c \in C$. Each edge $e \in E_T$ connects a vertex in P to one in C, or vice versa. Each process task $p \in P$ can have multiple incoming edges since it might receive data from multiple other process tasks. A process task can also have multiple outgoing edges to model the sending of data to multiple process tasks. The data itself is not directly sent to other processing tasks, but the transmission is modeled explicitly by communication tasks. Each communication task $c \in C$ has exactly one predecessor process task as the sender, since data is sent by exactly one sender. To allow multicast communication, each communication task can have multiple successor process tasks.
- The architecture is modeled as a directed graph $G_R(R, E_R)$. The vertices R represent resources such as processors, memories, or buses. The directed edges $E_R \subseteq R \times R$ indicate available communication connections between resources.

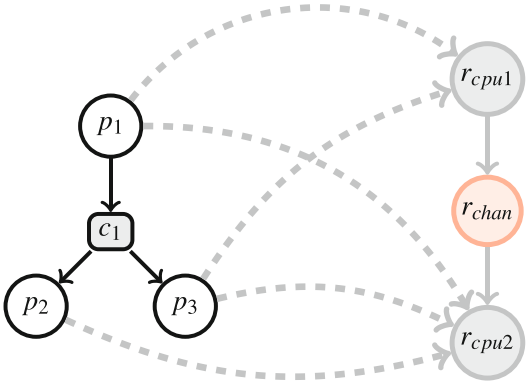


Fig. 7.1 Specification with the application graph G_T on *the left*, the architecture graph G_R on *the right*, and mapping edges depicted *dashed*

- The set of mapping edges E_M contains the mapping information for each process task. Each mapping edge $m = (p, r) \in E_M$ indicates a possible implementation/execution of process $p \in P$ on resource $r \in R$.

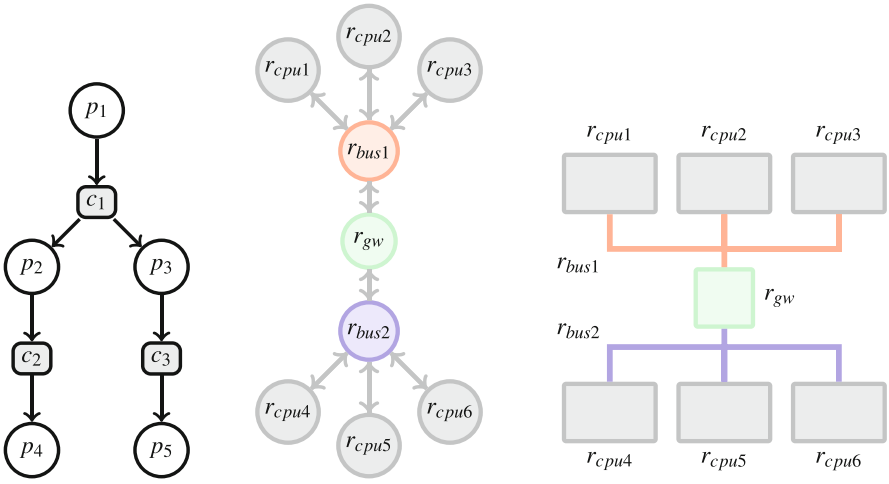
A simple specification including application graph, architecture graph, and mapping edges is given in Fig. 7.1: The application consists of three process tasks (p_1, p_2, and p_3) and one communication task (c_1) that distributes the data produced by p_1 to p_2 and p_3 in a multicast fashion. The architecture consists of two CPU resources capable of executing process tasks (r_{cpu1} and r_{cpu2}) that are connected via a channel (r_{chan}) that only allows a communication from r_{cpu1} to r_{cpu2} but not vice versa as specified by the directed edge $(r_{cpu1}, r_{cpu2}) \in E_R$. Moreover, mapping edges depict which process task can be executed on which resources, i.e., p_1 on r_{cpu1} and r_{cpu2}, p_2 on r_{cpu2}, and p_3 on r_{cpu1} and r_{cpu2}.

A more complex specification is given in Fig. 7.2: The application graph shown in Fig. 7.2a consists of five process tasks and three communication tasks. The architecture graph shown in Fig. 7.2b consists of six processors (CPUs) and two buses that are coupled via a gateway resource. For the sake of brevity and better visualization, we also show an architecture diagram in Fig. 7.2c where processors are depicted as rectangles and buses are depicted as edges between processors and (possibly) gateway components.

7.2.1.2 System-Level Synthesis

The introduced specification is the base for the following formulation of the system-level synthesis problem:

> System-level synthesis derives an *implementation* from a given specification by means of an *allocation* of resources, a *binding* of process tasks to allocated resources, a *routing* of communication tasks on a tree of allocated resources, and a *schedule* of tasks.



(a) $G_T(T, V_T)$ (b) $G_R(R, E_R)$ (c) architecture diagram

Fig. 7.2 A specification with (**a**) application graph G_T and (**b**) architecture graph G_R with mapping edges (not depicted) being $E_M = \{(p_1, r_{cpu1}), (p_1, r_{cpu2}), (p_2, r_{cpu5}), (p_3, r_{cpu2}), (p_3, r_{cpu4}), (p_4, r_{cpu4}), (p_4, r_{cpu6}), (p_5, r_{cpu1}), (p_5, r_{cpu3})\}$. A more compact representation of the modeled architecture as an architecture diagram is given in (**c**)

Formally, an implementation ω consists of the *allocation graph* G_α that is deduced from the architecture graph G_R, the *binding* E_β as a subset of E_M that describes the mapping of the process tasks to allocated resources, the *routing* γ that contains a directed routing graph $G_{\gamma,c}$ for each communication task $c \in C$, and the *schedule function* S. For an implementation to be *feasible*, several conditions have to be fulfilled by the allocation, binding, routing, and scheduling:

- The allocation is a directed graph $G_\alpha(\alpha, E_\alpha)$ that is an induced subgraph of the architecture graph G_R. The allocation contains all resources $r \in R$ that are selected for the current implementation. E_α describes the set of allocated communication connections that are induced from the graph G_R such that $e = (r, \tilde{r}) \in E_\alpha$ if and only if $r, \tilde{r} \in \alpha$.
- The binding $E_\beta \subseteq E_M$ describes the mapping of the process tasks to allocated resources. Here, the following requirements must be fulfilled:
  - Each process task $p \in P$ of the application is bound to exactly one resource:

$$\forall p \in P : |\{m | m = (p, r) \in E_\beta\}| = 1 \tag{7.1}$$

    where $|\cdot|$ denotes the cardinality.
  - Each process task $p \in P$ can only be bound to an allocated resource:

$$\forall m = (p, r) \in E_\beta : r \in \alpha \tag{7.2}$$

- Each communication task $c \in C$ is routed on a tree $G_{\gamma,c} = (R_{\gamma,c}, E_{\gamma,c})$. The routing must be performed such that the following conditions are satisfied:
  - The directed tree $G_{\gamma,c}$ is a connected subgraph of the allocation G_α such that

$$R_{\gamma,c} \subseteq \alpha \text{ and } E_{\gamma,c} \subseteq E_\alpha. \tag{7.3}$$

  - For each communication task $c \in C$, the root of $G_{\gamma,c}$ has to equal the resource on which the predecessor sender process task $p \in P$ is bound:

$$\forall (p,c) \in E_T, m = (p,r) \in E_\beta : |\{e | e = (\tilde{r}, r) \in E_{\gamma,c}\}| = 0 \tag{7.4}$$

    Here, r is the resource to which the sender process task $p \in P$ is bound and by requiring that the number of incoming edges to this node is 0, the node is ensured to be the routing tree's root.
  - For each communication task $c \in C$, $R_{\gamma,c}$ must contain all resources on which any successor process task $p \in P$ is bound:

$$\forall (c,p) \in E_T, m = (p,r) \in E_\beta : r \in R_{\gamma,c} \tag{7.5}$$

- On each resource, different scheduler types may be present to schedule tasks bound to them. Here, a general scheduling approach is considered which assigns each mapping of a (process and communication) task a priority:

$$S : E_M \cup C \rightarrow \{1, \ldots, |T| + |C|\}. \tag{7.6}$$

  In case resources either execute process tasks or route communication tasks, the number of required priorities decreases to $\max(|T|, |C|)$.

  It is then the responsibility of the scheduler to consider the assigned priorities. However, the following two conditions typically have to be fulfilled:
  - Process task priorities are unique per resource $r \in R$:

$$\forall r \in R, m = (t,r), m' = (t',r) \in E_\beta, m \neq m' : S(m) \neq S(m') \tag{7.7}$$

  - Communication task priorities are unique (since they may share several allocated resources on their route):

$$\forall r \in R, c, c' \in C, c \neq c' : S(c) \neq S(c') \tag{7.8}$$

Two implementations for the simple specification given in Fig. 7.1 are shown in Fig. 7.3. The implementation depicted on the left is infeasible since c_1 cannot be routed to p_3. The implementation depicted on the right adheres to all requirements and is, thus, a feasible implementation. Moreover, a feasible implementation for the more complex specification from Fig. 7.2 is given in Fig. 7.4.

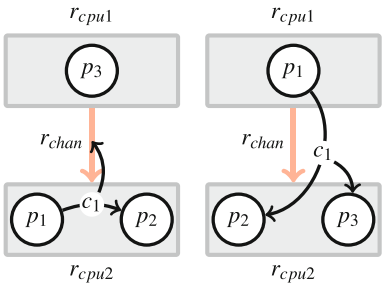


Fig. 7.3 Two implementations of the specification in Fig. 7.2. Shown are two allocated resources r_{cpu1} and r_{cpu2} as *rectangles* and the unidirectional communication channel r_{chan} as a *directed edge* between them. The left implementation is infeasible: Because of the unidirectional communication channel, there exists no connected subgraph to route the communication task c_1 to receiver p_3. The right implementation is feasible

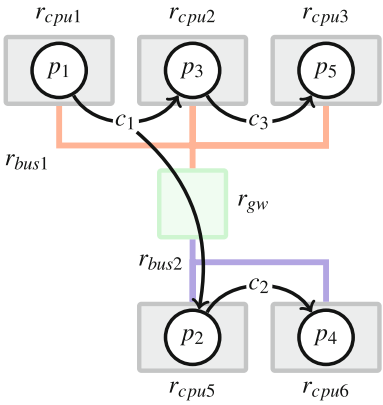


Fig. 7.4 An implementation for the specification in Fig. 7.2. Illustrated is the allocation G_α, the binding E_β of process tasks, and the routing γ of communication tasks. All routings are performed within multiple hops using the available bi-directional buses r_{bus1}, r_{bus2} and the gateway r_{gw}. The communication c_1 is of type multicast

The outlined system-level synthesis problem is typically represented by means of the Y-chart [5] where the separation of application and architecture resulting in an implementation forms a Y as depicted in Fig. 7.5. In general, system-level synthesis shall obviously only deliver feasible implementations. Yet, our aim is to search for implementations that are optimized with respect to multiple and even conflicting design objectives. For this purpose, the Y-chart approach is extended in [1] and later in [12] by a DSE phase that can be seen as an optimization in order to obtain high-quality implementations. This directly brings us to the core topic of this chapter: How to efficiently perform a DSE which has to solve the complex system-level synthesis problem for each considered implementation? Before we come to the introduction of the hybrid optimization technique, we briefly review optimization techniques that are suitable for such kind of problems.

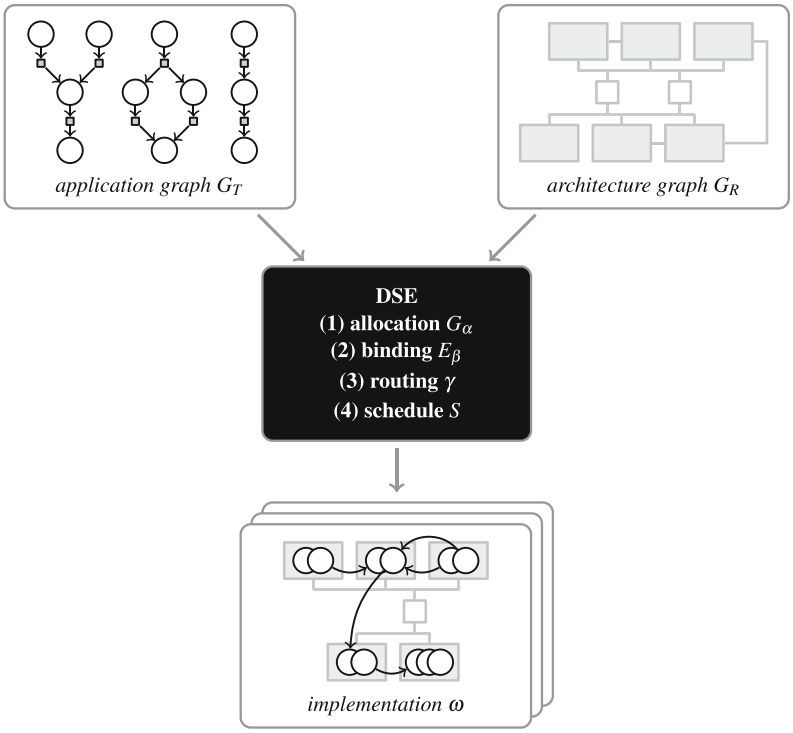


Fig. 7.5 Illustration of the Y-chart approach: An implementation shall be synthesized from a given application graph and architecture graph. For each implementation that is considered during DSE, resource allocation, process task binding, communication task routing, and task scheduling have to be determined with the overall goal to determine a set of Pareto-optimal implementations

7.2.2 Constrained Combinatorial Optimization

As we have seen, whether an implementation that is deduced from a specification is feasible requires it to fulfill the introduced constraints. Also, we can recognize that allocation, binding, routing, and scheduling are all design steps that basically solve combinatorial problems of assigning tasks to resources or priorities to tasks. Thus, we can conclude that system-level synthesis as introduced falls in the class of *constrained combinatorial problems*:

Definition 1 (Constrained Combinatorial Problem).

$$\begin{aligned} &\textit{minimize } f(\mathbf{x}) \\ &\textit{subject to:} \\ &\quad a_i(\mathbf{x}) \leq b_i, \forall i \in \{1, \ldots, q\} \text{ with } b_i \in \mathbb{Z} \end{aligned}$$

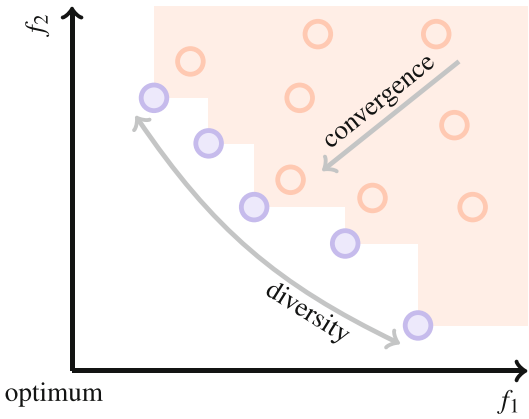


Fig. 7.6 A two-dimensional objective space: Pareto-optimal implementations are depicted *light blue* while dominated implementations are depicted *light red*. The area which is dominated by Pareto-optimal implementations is depicted *light red* as well. Multi-objective optimization approaches try to achieve high convergence as well as high diversity among the found implementations

In system-level synthesis, the objective function f is typically multi-dimensional to consider multiple objective functions such as area, power, or timeliness that can, in particular, be non-linear. Note that in multi-objective optimization problems, there is generally not only one global optimum, but also a set of so-called *Pareto-optimal implementations* is derived with each Pareto-optimal implementation being better in at least one objective when compared to each other feasible implementation. In multi-objective optimization, the notion of one implementation being better than another is given by the concept of *dominance* ($\prec$), i.e., one implementation dominates another if it is better in at least one design objective. Figure 7.6 visualizes this in the objective space with two design objectives f_1 and f_2. Pareto-optimal implementations are depicted light blue and dominated implementations light red. Also, the areas in the objective space that are dominated by a Pareto-optimal implementation are depicted. An indicator for the quality of a multi-objective optimization approach is *convergence*, i.e., how close are the found implementations to the front or Pareto-optimal implementations, and *diversity*, i.e., how well distributed are the implementations in the *objective space*.

For this chapter, focus is not put on the handling of multiple objectives. Instead, the main problem addressed arises from the notion of the *feasible search space* $X_\mathrm{f} \subseteq X$. In the general form, the search space is constrained by q so-called *constraint functions* a imposed on an implementation, formulated as inequalities $a_i(\mathbf{x}) \leq b_i$. The effect of these restrictions is sketched in Fig. 7.7: While every point in the search space of an unconstrained combinatorial problem is feasible, the search space $X_\mathrm{f} \subseteq X$ may contain significantly less or – in the extreme case – even no feasible implementation at all.

In order to solve this problem efficiently, the search space and the types of constraints will be restricted in the following: The search space $X = \{0,1\}^n$ is encoded as a set of two Boolean vectors. Moreover, the constraints are restricted to a single matrix inequation as follows:

$$A\mathbf{x} \leq b \text{ with } \mathbf{x} \in \{0,1\}^n, A \in \mathbb{Z}^{m,n}, b \in \mathbb{Z}^m \tag{7.9}$$

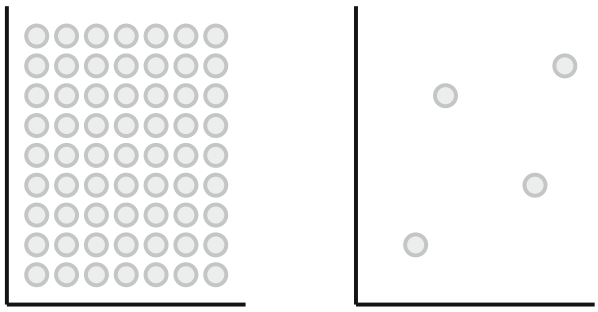


Fig. 7.7 A visualization of the search space of a combinatorial problem (*left*) and a constrained combinatorial problem (*right*) when projecting the search space $X = \{0, 1\}^n$ onto two dimensions: In the unconstrained case, every point in the search space is a feasible implementation while in the constrained case, large areas in the search space might not contain any feasible implementation

As can be seen, the constraints given by $A\mathbf{x} \leq b$ have to be linear or linearizable. In the two main parts of this chapter, we will first discuss that the introduced constraints for allocation, binding, routing, and scheduling can be linearized and, thus, directly be included in the search space. Afterward, we will also introduce how to take care of constraints that cannot be linearized.

As outlined in ▸ Chap. 6, "Optimization Strategies in Design Space Exploration", metaheuristic optimization techniques have become state-of-the-art to solve several problems from the area of hardware/software codesign. This is mainly due to their ability to consider multiple conflicting and even non-linear design objectives. In contrast, exact approaches like Integer Linear Programs (ILPs) require the objective function to be linear and multiple objectives are – in general (see [15]) – not supported. However, applying metaheuristic optimization techniques to constrained combinatorial problems as given by system-level synthesis raises a significant problem: How to determine the set of Pareto-optimal feasible implementations and – in case of really stringent constraints – how to even find one single feasible implementation?

The basic idea behind the relevant metaheuristic approaches is to *vary* selected (high-quality) implementations in an iterative loop and to keep the best found so far in an *archive* $\mathbb{A}$ of non-dominated implementations. In every iteration, varied implementations, e.g., ω, are compared to the ones from the archive, e.g., $\tilde{\omega}$, and – also depending on the concrete metaheuristic optimization technique – either dropped in case they are dominated ($\prec$) by implementations from the archive, i.e., $\omega \prec \tilde{\omega}$, or the archive is updated with the new implementations. This way, metaheuristics gradually but efficiently search for the best implementations. This principle is depicted in Fig. 7.8. However, in the presence of stringent constraints, Fig. 7.9 (left) outlines the effect of variation. The next feasible implementation may be out of reach and since all surrounding implementations are infeasible, only a very slow convergence toward the optimal implementations is achieved. In some cases, it might even occur that not even a single feasible implementation is found.

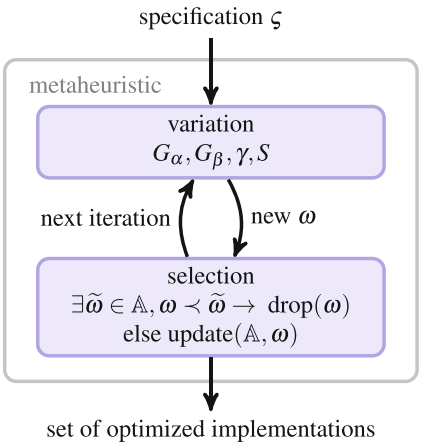


Fig. 7.8 Principle of metaheuristic optimization approaches for hardware/software codesign: Given a specification, the heuristic performs an iterative optimization loop where it varies the allocation, binding, routing, and schedule to explore implementations and selects which implementations (**a**) are to be dropped since they are dominated by implementations from an archive, (**b**) update the archive in case they dominate implementations in the archive, and (**c**) are promising candidates for variation in the next iteration. At the end, a set of optimized (near Pareto-optimal) implementations is the output

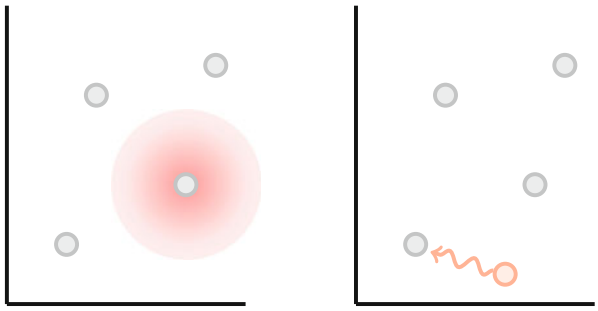


Fig. 7.9 Varying a feasible implementation as a common concept of most metaheuristic optimization techniques may only result in neighboring implementations that are all infeasible (*left*). In the presence of a repair strategy, an infeasible implementation is modified to – if possible – become a feasible implementation (*right*)

As a remedy, *constraint-handling techniques* have been successfully developed to apply metaheuristic optimization techniques to constrained combinatorial problems; see [2] for a comprehensive overview. Here, we will just introduce two main concepts: *Penalty functions* and *repair strategies*. The idea of penalty functions [27] is to transform the constrained problem into an unconstrained problem by deteriorating the original objective function by a penalty function. The amount of penalization depends on the violation of constraints. A similar idea is to leave the original objective function as is and add constraints as additional objectives [11], e.g., minimizing the number of violated constraints. The drawbacks

of these approaches are the increased complexity of the problem by additional objectives and a slow convergence if the feasible region is relatively small compared to the entire search space.

For some combinatorial problems, repair algorithms or at least repair heuristics are available that restore the feasibility of the implementation by applying certain modifications. One well-known problem where such a solution exists is the *0/1 Knapsack Problem* [32] where an infeasible implementation can be repaired by removing items from the knapsack. The idea of the repair strategy is depicted in Fig. 7.9 (right). Since the introduced system-level synthesis problem is NP-complete, also a repair heuristic is, in general, NP-complete and, thus, does not offer a conclusive alternative for our outlined problem.

The rest of this chapter introduces a solution to this problem by means of a hybrid optimization approach.

7.3 Hybrid Optimization

This section introduces a hybrid optimization technique for system-level DSE of embedded systems. First, the key idea of the hybrid optimization technique termed *SAT decoding* – the combination of a metaheuristic with a backtracking-based search algorithm (solver) to consider only feasible implementations during DSE – is presented. Afterward, the main required ingredients to realize such an approach, i.e., (a) the branching strategy of the solver and (b) the formulation of the system-level synthesis problem by means of pseudo-Boolean constraints, are explained.

7.3.1 SAT Decoding: The Key Idea

In the literature, one can find various hybrid optimization approaches that combine heuristic algorithms with exact approaches, also for combinatorial problems from diverse domains; cf. [23] for an overview. The approach discussed in the chapter at hand termed *SAT decoding* [16] falls into the category of *integrative* hybrid optimization approaches. Here, a metaheuristic algorithm is responsible to control the overall optimization procedure: It controls the optimization loop and selects implementations based on their multiple and even non-linear design objectives. In contrast to standard metaheuristics, it does not directly vary the implementation (i.e., allocation, binding, routing, and scheduling). Instead, it integrates a solver – in our case a *Pseudo-Boolean* (PB) solver – that is only responsible to gather feasible implementations, but does not perform any optimization by itself. Following is the key idea of SAT decoding:

> In SAT decoding, instead of varying the implementation directly, the metaheuristic varies the *branching strategy* of the backtracking-based solver. This way, only feasible implementations are obtained and are evaluated during design space exploration.

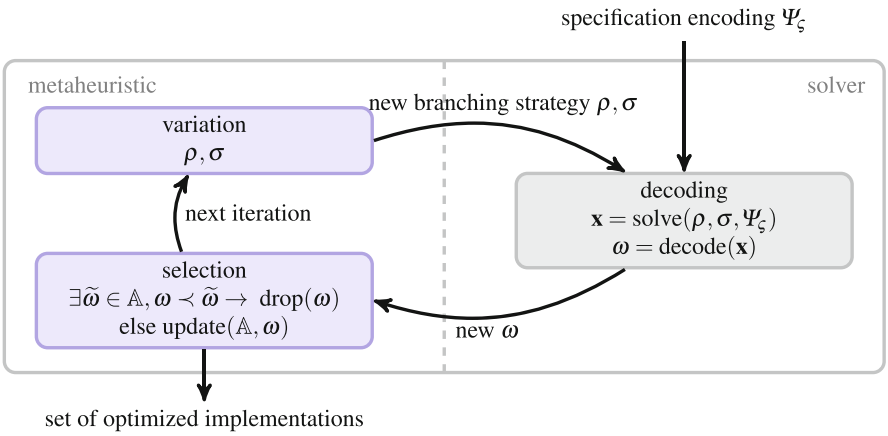


Fig. 7.10 Principle of the SAT-decoding approach: The metaheuristic does not vary the implementation directly but rather varies the parameters σ, ρ of the branching strategy of the employed solver. The solver takes an encoding Ψ_ς of the specification ς together with the current branching strategy and determines a feasible implementation. Each feasible implementation is then delivered to the selection step of the metaheuristic. Note that this hybrid optimization approach only derives feasible implementations to the metaheuristic, circumventing the outlined problems of other state-of-the-art design space exploration approaches for system-level synthesis

Figure 7.10 shows the idea and the resulting integrative hybrid optimization approach. The key feature of this approach is that it combines the advantages of the metaheuristic, i.e., being able to consider multiple and non-linear design objectives, with those of the solver, i.e., only obtaining feasible implementations. After this informal introduction of the approach, each fundamental ingredient is presented in a more detailed fashion in the following subsection.

7.3.2 Solver

To understand how the metaheuristic can actually control the solver to explore the whole diversity of different implementations, we first need to understand how the solver finds feasible implementations – or better – how it solves given search problems in general. Thus, this subsection first introduces the so-called *Pseudo-Boolean* (PB) problem and then shows how most existing solvers approach them by means of the *Davis-Putnam-Logemann-Loveland* (DPLL) backtracking algorithm.

In general, the task of a PB solver is to find a *variable assignment* $\mathbf{x}$ that satisfies a set of linear constraints, i.e., $\mathbf{x} \in X_\text{f}$ as formulated in Equation (7.9). Constraints that are given as linear inequalities and Boolean variables with integer coefficients are known as *PB constraints*. Any ILP solver is also capable of solving PB problems. But, specialized *PB solvers* tend to outrun ILP solvers on Pseudo-Boolean problems because they are based on efficient backtracking algorithms. In fact, many PB solvers are extended Boolean Satisfiability (SAT) solvers with the capability to handle PB constraints and rely on the Davis-Putnam-Logemann-

Loveland (DPLL) algorithm [3]. To comprehend the SAT-decoding technique requires an understanding of the main concept of the DPLL algorithm which is outlined in Algorithm 2. The algorithm starts with a set of completely unassigned

Algorithm 2 DPLL backtracking algorithm

```
1:  procedure SOLVE(σ, ρ)
2:      while true do
3:          branch(σ, ρ)
4:          if CONFLICT then
5:              backtrack()
6:          else if SATISFIED then
7:              return x
8:          end if
9:      end while
10: end procedure
```

variables. Then, the operation branch(σ, ρ) selects an unassigned variable $\mathbf{x}_i$ and assigns it the value 0 or 1 (Line 3). Which variable is selected and which value (0 or 1) is assigned is called *branching strategy*. The branching strategy – which is of key importance for the SAT-decoding approach – is guided by the vectors $\sigma \in \{0,1\}^n$ and $\rho \in \mathbb{R}^n$: Unassigned variables $\mathbf{x}_i$ with the highest value ρ_i are prioritized and set to the value σ_i. Of course, as is common in all backtracking-based solvers, arising conflicts are recognized and resolved (Line 4). A conflict is recognized if any constraint is not satisfiable anymore and *backtracking* is triggered (Line 5). Backtracking means that variable assignments made before are reverted. When all variables have a variable assignment and no conflict occurs (Line 6), then the variable assignment is a feasible solution to the specified problem and returned (Line 7). The majority of the state-of-the-art PB solvers like SAT4J [13] are based on the DPLL algorithm.

Knowing the algorithm of the solver, we can draw two important conclusions: First, we can control which variable assignment, i.e., which feasible implementation, is delivered by the solver by varying the two vectors σ and ρ of the branching strategy. Thus, the *search space* of the metaheuristic in SAT decoding is not the allocation G_α, the binding E_β, the routing γ, and the schedule S, but is solely given by the two vectors σ and ρ of the solver's branching strategy. Second, we have to find an *encoding* Ψ_ς of a specification ς and the introduced system-level synthesis problem by means of pseudo-Boolean constraints such that each feasible variable assignment $\mathbf{x}$ represents a feasible implementation of a given specification.

7.3.3 Pseudo-Boolean Encoding of Allocation, Binding, Routing, and Scheduling

In the following, we present a pseudo-Boolean encoding Ψ_ς of a specification ς and the system-level synthesis problem such that a solution $\mathbf{x} \in \{0,1\}^n$ corresponds to a *feasible* implementation ω according to Equations (7.1), (7.2), (7.3), (7.4), (7.5),

(7.6), (7.7), and (7.8). First, we introduce the required Boolean variables used to formulate the linear constraints:

r
: A Boolean variable for each resource $r \in R$. It indicates whether the resource is allocated $r \in \alpha$ (1) or not (0).

m
: A Boolean variable for each mapping edge $m \in E_M$. It indicates whether the mapping edge is part of the binding, i.e., $m \in E_\beta$ (1) or not (0).

$\mathbf{c_r}$
: A Boolean variable for each communication task $c \in C$ and resource $r \in R$. It indicates whether the communication task c is routed over the resource r (1) or not (0)

$\mathbf{c_{r,\tau}}$
: A Boolean variable for each communication $c \in C$ and resource $r \in R$. It indicates at which communication step $\tau \in \mathcal{T} = \{1, .., |\mathcal{T}|\}$ a communication is routed over the resource. Note that communication tasks are propagated in steps or hops, respectively.

With these variables, we can formulate the linear constraints that encode all introduced requirements in Equations (7.1), (7.2), (7.3), (7.4), (7.5), (7.6), (7.7), and (7.8) for a feasible implementation. First, we start with the linear constraints regarding allocation, binding, and routing:
$\forall p \in P$:

$$\sum_{m=(p,r)\in E_M} \mathbf{m} = 1 \tag{7.10}$$

$\forall m = (p, r) \in E_M$:

$$\mathbf{r} - \mathbf{m} \geq 0 \tag{7.11}$$

The constraints in Equations (7.10) and (7.11) ensure that each task is bound to exactly one resource (cf. Equation (7.1)) and that this resource is allocated (cf. Equation (7.2)). Exemplarily, we show the constraints that would result from Equations (7.10) and (7.11) for the simple specification given in Fig. 7.1:

$$\begin{aligned}
\mathbf{m}_{p_1,r_{cpu1}} + \mathbf{m}_{p_1,r_{cpu2}} &= 1 \\
\mathbf{m}_{p_2,r_{cpu2}} &= 1 \\
\mathbf{m}_{p_3,r_{cpu1}} + \mathbf{m}_{p_3,r_{cpu2}} &= 1 \\
\mathbf{r}_{cpu1} - \mathbf{m}_{p_1,r_{cpu1}} &\geq 0 \\
\mathbf{r}_{cpu1} - \mathbf{m}_{p_3,r_{cpu1}} &\geq 0
\end{aligned}$$

$$\mathbf{r}_{cpu2} - \mathbf{m}_{p_1,r_{cpu2}} \geq 0$$
$$\mathbf{r}_{cpu2} - \mathbf{m}_{p_2,r_{cpu2}} \geq 0$$
$$\mathbf{r}_{cpu2} - \mathbf{m}_{p_3,r_{cpu2}} \geq 0$$

These first constraints cover the major requirements regarding process task binding and the allocation of the resources to execute them. The following constraints cover the aspect of routing data from sender to receiver and, thus, address Equations (7.3), (7.4), and (7.5):

$\forall c \in C, r \in R, (c, p) \in E_T, m = (p, r) \in E_M$:

$$\mathbf{c_r} - \mathbf{m} = 0 \tag{7.12}$$

$\forall c \in C$:

$$\sum_{r \in R} \mathbf{c_{r,1}} = 1 \tag{7.13}$$

$\forall c \in C, r \in R, (p, c) \in E_T, m = (p, r) \in E_M$:

$$\mathbf{m} - \mathbf{c_{r,1}} = 0 \tag{7.14}$$

Equation (7.12) ensures that a communication task c is routed to each resource a succeeding (receiving) process task is mapped to (cf. Equation (7.5)). Analogously, the constraints in Equations (7.13) and (7.14) ensure a communication task's root is the resource that the preceding (sending) process task is mapped to (cf. Equation (7.4)). Having the very basic routing constraints formulated, we need further constraints to precisely formulate what makes a route between source and multiple receivers feasible. First, we ensure that each communication task can only be routed on allocated resources by Equation (7.15):

$\forall c \in C, r \in R$:

$$\mathbf{r} - \mathbf{c_r} \geq 0 \tag{7.15}$$

Additionally, Equation (7.16) ensures that a communication task may be routed only between adjacent resources in one communication step:

$\forall c \in C, r \in R, \tau = \{2, .., |\mathcal{T}|\}$:

$$\left(\sum_{\tilde{r} \in R, e=(\tilde{r},r) \in E_R} \mathbf{c_{\tilde{r},\tau}} \right) - \mathbf{c_{r,\tau+1}} \geq 0 \tag{7.16}$$

It is finally required that a communication task is assigned a communication step τ if it is considered to be routed over a resource which is achieved by Equations (7.17) and (7.18):

$\forall c \in C, r \in R$:

$$\left(\sum_{\tau \in \mathscr{T}} \mathbf{c_{r,\tau}} \right) - \mathbf{c_r} \geq 0 \tag{7.17}$$

$\forall c \in C, r \in R, \tau \in \mathscr{T}$:

$$\mathbf{c_r} - \mathbf{c_{r,\tau}} \geq 0 \tag{7.18}$$

Here, Equation (7.17) ensures that if $\mathbf{c_r}$ is set to 1, the sum of $\mathbf{c_{r,\tau}}$ variables is greater zero which means that if the message is routed on the resource, a respective time step has to be assigned. On the other hand, Equation (7.18) ensures that no $\mathbf{c_{r,\tau}}$ variable can be set to 1 unless $\mathbf{c_r}$ is set to 1 as well.

The introduced Equations (7.10), (7.11), (7.12), (7.13), (7.14), (7.15), (7.16), (7.17), and (7.18) are suitable to ensure a feasible allocation, binding, and routing and, thus, a feasible implementation in case scheduling is of no concern.

Further constraints may be added to enhance the obtained feasible implementations. First, a natural enhancement to a feasible route is to require it to be free of (redundant) cycles. The satisfaction of Equation (7.19) avoids cycles in a route by ensuring that a communication task can pass a resource at most once:
$\forall c \in C, r \in R$:

$$\sum_{\tau \in \mathscr{T}} \mathbf{c_{r,\tau}} \leq 1 \tag{7.19}$$

Second, an implementation benefits from not containing unused (redundant) resources as they might affect design objectives such as cost, area, or power consumption. To eliminate unused resources from the allocation, Equation (7.20) ensures that a resource is only allocated if at least one process or communication task is bound to or routed over it:
$\forall r \in R$:

$$\left(\sum_{c \in C \wedge r \in R} \mathbf{c_r} \right) + \left(\sum_{m=(p,r) \in E_M} \mathbf{m} \right) - \mathbf{r} \geq 0 \tag{7.20}$$

The pseudo-Boolean encoding presented so far covers the allocation, binding, and routing.

As outlined in the definition of the system-level synthesis problem, we finally want to support generic scheduling constraints by means of assigning priorities. Thus, we now introduce a pseudo-Boolean encoding for task and communication priorities; cf. Equations (7.7) and (7.8). Here, we again need to introduce Boolean variables:

$\mathbf{s}_{p,\tilde{p}}$
: A Boolean variable that indicates whether process task $p \in P$ has a higher priority than task $\tilde{p} \in P$ (1) or not (0).

$\mathbf{s}_{c,\tilde{c}}$
: A Boolean variable that indicates whether communication task $c \in C$ has a higher priority than task $\tilde{c} \in C$ (1) or not (0).

The following constraints ensure that priorities are assigned properly. We first define the priority assignment function for process tasks which assigns correct priorities within tasks bound to the same resource (cf. Equation (7.7)):
$\forall r \in R, (p,r), (\tilde{p},r) \in E_M, p \neq \tilde{p}$:

$$\mathbf{s}_{p,\tilde{p}} + \overline{\mathbf{s}_{\tilde{p},p}} = 1 \tag{7.21}$$

Equation (7.21) states that if process task p has a higher priority than task $\tilde{p}$ ($\mathbf{s}_{p,\tilde{p}} = 1$), it has to be ensured that task $\tilde{p}$ has a lower priority than task p ($\mathbf{s}_{\tilde{p},p} = 0$). Now, we also have to ensure transitivity, i.e., task p has a higher priority than task $\tilde{p}$ and task $\tilde{p}$ than task $\hat{p}$. It also has to hold that task p has higher priority than task $\hat{p}$ which is ensured by Equations (7.22) and (7.23):
$\forall r \in R, (p,r), (\tilde{p},r), (\hat{p},r) \in E_M, p \neq \tilde{p} \neq \hat{p}$:

$$\mathbf{s}_{p,\tilde{p}} + \mathbf{s}_{\tilde{p},\hat{p}} + \overline{\mathbf{s}_{p,\hat{p}}} \leq 2 \tag{7.22}$$

$$\overline{\mathbf{s}_{p,\tilde{p}}} + \overline{\mathbf{s}_{\tilde{p},\hat{p}}} + \mathbf{s}_{p,\hat{p}} \leq 2 \tag{7.23}$$

Exactly the same requirements as ensured by Equations (7.21), (7.22) and (7.23) are now imposed on the communication tasks as well. The only difference is that – since communication tasks might share multiple buses and other communication resources – we apply a global priority assignment; cf. Equation (7.8):
$\forall c, \tilde{c} \in C, c \neq \tilde{c}$:

$$\mathbf{s}_{c,\tilde{c}} + \overline{\mathbf{s}_{\tilde{c},c}} = 1 \tag{7.24}$$

$\forall c, \tilde{c}, \hat{c} \in C, c \neq \tilde{c} \neq \hat{c}$:

$$\mathbf{s}_{c,\tilde{c}} + \mathbf{s}_{\tilde{c},\hat{c}} + \overline{\mathbf{s}_{c,\hat{c}}} \leq 2 \tag{7.25}$$

$$\overline{\mathbf{s}_{c,\tilde{c}}} + \overline{\mathbf{s}_{\tilde{c},\hat{c}}} + \mathbf{s}_{c,\hat{c}} \leq 2 \tag{7.26}$$

With the above-given constraints, a unique priority assignment is achieved.

From the introduced encoding Ψ_ς, a simple *decode* function as shown in Fig. 7.10 can be defined that derives the concrete implementation $\omega = \text{decode}(\mathbf{x}) = (G_\alpha, G_\beta, \gamma, S)$ from the phase of the Boolean variables in $\mathbf{x}$. This is also depicted in the decoding step shown in Fig. 7.10.

As outlined, the search space of SAT decoding consists solely of the two vectors σ and ρ of the branching strategy. Given the mentioned rules to determine an

encoding Ψ_ς, the search space can be seamlessly derived by providing one entry in σ and ρ for each variable required for the encoding Ψ_ς. This completes the introduction of the basic ingredients of the SAT-decoding approach.

7.4 Satisfiability Modulo Theories During Decoding

The introduced SAT-decoding approach is capable of restricting the search space to feasible implementations with respect to given linear constraints (cf. Definition 1) only. However, in the area of hardware/software codesign, several objectives can – typically – not be linearized in a sound fashion. Two prominent examples are *timeliness* which requires to analyze the interference of process and communication tasks on shared resources and *reliability* which is a probabilistic and combinatorial problem itself that has to consider which combination of faults in tasks or resources results in the system to fail. As can be imagined, transforming such complex behaviors and interactions into a combination of linear constraints may come at significant over-approximations or even result in practically useless results. In this section, we therefore introduce the key idea how to also consider non-linear constraints, followed by a formal definition of the SMT decoding technique. Afterward, different schemes how to learn which solutions are infeasible with respect to a set of non-linear constraints are discussed.

7.4.1 SMT Decoding: The Key Idea

The solution to the problem of considering non-linear constraints is inspired by the concept of *Satisfiability Modulo Theories* (SMT); cf. [4]. Without aiming for a complete and thorough introduction of SMT, the basic idea of SMT relevant for hardware/software codesign is that it checks the satisfiability of a logical formula over one or more *background theories*. The concept of *SMT decoding* [24] is depicted in Fig. 7.11. We again use the pseudo-Boolean encoding Ψ_ς that considers a set of linear constraints as introduced in the last section. We now hand over an implementation ω delivered by the solver to one or several background theories, each of which decides whether the implementation is feasible (isFeasible(ω)) for a set of non-linear constraints as well. In the context of hardware/software codesign, such a background theory could, for example, be a formal timing analysis (cf. ▸ Chap. 23, "CPA: Compositional Performance Analysis") or a timing simulation (cf. ▸ Chap. 19, "Host-Compiled Simulation") that can decide whether the delay of an implementation meets a certain deadline. So in particular, we *couple* any external analysis technique for any interesting system constraint as a background theory to the introduced solver. In case the variable assignment does not fulfill those constraints that are checked by the background theory, the solver will be told to consider this variable assignment as infeasible (although it initially appeared to be feasible considering only the set of linear constraints). This way, the solver

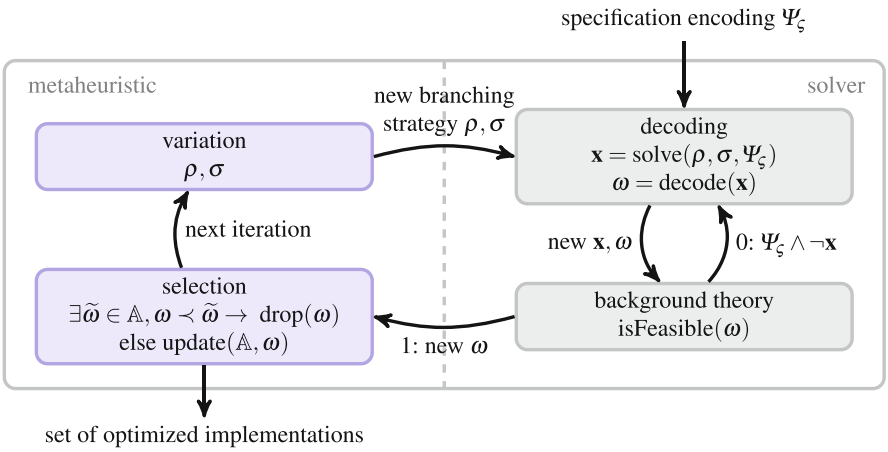


Fig. 7.11 Principle of the SMT decoding approach: The solver takes an encoding of the specification together with the current branching strategy and determines a feasible implementation ω with respect to a set of linear constraints. The background theory then checks the delivered implementation for feasibility with respect to a set of non-linear constraints. In case it is infeasible (0), the respective variable assignment is excluded in the solver and the solver is asked for a new implementation. This way, the solver *learns* over time which variable assignments violate a set of non-linear constraints

basically *learns* which variable assignments are infeasible with respect to a set of non-functional constraints over time.

Consider again the simple specification given in Fig. 7.1 and the implementation that is feasible with respect to the linear constraints defined by system-level synthesis depicted in Fig. 7.3 on the right. Assume we are interested in formulating a set of additional constraints on timeliness of computed results. For example, we formulate a deadline for the execution of the application of the simple system shown in Fig. 7.1. Here, we employ a formal timing analysis approach as our background theory to check whether the latency of each implementation meets the specified deadline. As indicated in Fig. 7.12 on the left, the implementation violates the deadline because p_2 and p_3 have to be executed sequentially on r_{cpu2} (depicted also in the Gantt chart at the bottom left). Thus, the implementation is infeasible with respect to timeliness and this information has to be propagated to the solver. This is achieved by combining the specification encoding Ψ_ς with an encoding $\mathbf{x}_\omega$ of this implementation such that this implementation is *not* feasible anymore. In the concrete example – and for the sake of brevity only considering the mapping variables – this results in $\Psi_\varsigma \wedge \neg(\mathbf{m}_{p_1,r_{cpu1}} \wedge \mathbf{m}_{p_2,r_{cpu2}} \wedge \mathbf{m}_{p_3,r_{cpu2}})$. This can be achieved by a Boolean conjunction of the specification encoding with the negated implementation encoding, i.e., $\Psi_\varsigma \wedge \neg \mathbf{x}_\omega$. Figure 7.12 on the right shows an implementation that adheres to both linear constraints and the non-linear constraint on timeliness – tasks p_2 and p_3 can be executed in parallel on r_{cpu2} and r_{cpu1}, respectively.

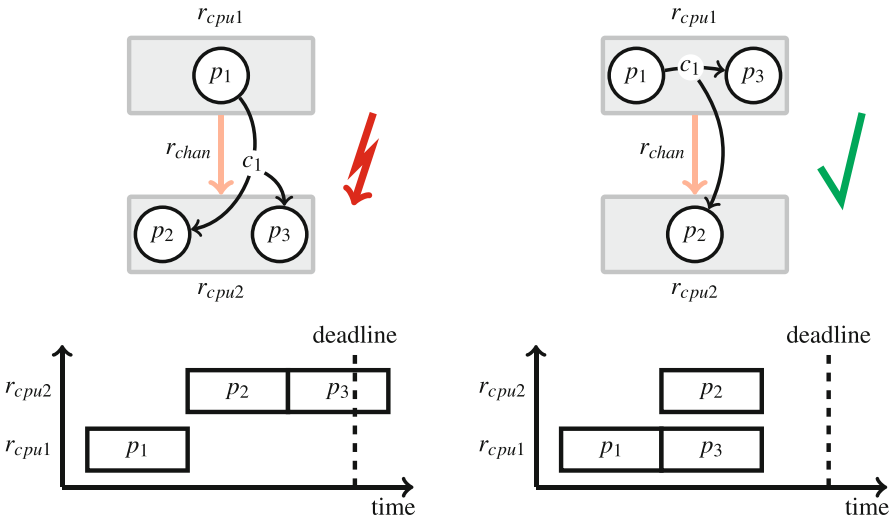


Fig. 7.12 Two implementations for the specification in Fig. 7.1 that are both feasible with respect to the introduced set of linear constraints. Yet, the implementation shown on *the left* does not meet a specified deadline because p_2 and p_3 have to run sequentially on r_{cpu2} after receiving the data from p_1 (see Gantt chart at *bottom left* obtained from the timing analysis). Thus, the implementation is infeasible with respect to a constraint on timeliness. On *the right*, the deadline can be met because p_2 and p_3 can run in parallel once both received the data from p_1 (see Gantt chart at *bottom right*)

This way, the key idea of SAT decoding – the restriction of the search space to feasible implementations only – can be extended to non-linear constraints as well by means of SMT decoding. In the following subsection, we give a formal definition of SMT decoding that completes the informal introduction given so far. Afterward, we introduce at which points of the solving process a feasibility check by the background theory can be applied, resulting in different *learning schemes* of the solver.

7.4.2 SMT Decoding Formulation

Let $\Omega_\mathrm{f} \subseteq \Omega$ denote the subset of feasible implementations of all implementations Ω. Those feasible implementations $\Omega_\mathrm{f} = \Omega_L \cap \Omega_N$ are given by the cut set of those implementations Ω_L that are feasible with respect to the set of linear constraints and implementations Ω_N that are feasible with respect to non-linear constraints. What we know from the previous section is that we can derive a pseudo-Boolean encoding Ψ_ς for our system-level synthesis problem that delivers Ω_L. Our aim is to derive an encoding Ψ_f for all feasible implementations Ω_f which would be given as $\Psi_\mathrm{f} = \Psi_\varsigma \wedge \Psi_N$. However, Ω_N cannot be converted to a respective Pseudo-Boolean (PB) encoding Ψ_N because we cannot linearize those constraints in a sound fashion. From this problem, we can formalize the key idea of SMT decoding:

In SMT decoding, an encoding Ψ_N for the set of implementations Ω_N that are feasible with respect to a set of non-linear constraints is derived by iteratively *learning* the implementations $\overline{\Psi_N}$ that are *infeasible* using one or several background theories. Whenever a variable assignment $\mathbf{x}$ is considered infeasible by the background theory, it is added to $\overline{\Psi_N}$ via $\overline{\Psi_N}^{i+1} := \overline{\Psi_N}^{i} \vee \mathbf{x}$. SMT decoding is, thus, capable of deriving Ψ_f via $\Psi_\mathrm{f} = \Psi_\varsigma \wedge \neg\overline{\Psi_N}$, i.e., the conjunction of those implementations that are feasible with respect to a set of linear constraints and *not* those that do violate any non-linear constraint.

Since SMT decoding aims at learning $\overline{\Psi_N}$ iteratively and does not require a closed-form representation, any analysis technique can be employed to determine whether an implementation is feasible or not. This results in a great flexibility and applicability of SMT decoding to various aspects and problems from the area of hardware/software codesign.

Note that this technique is even capable of covering a delicate corner case: In case no feasible implementation exists, i.e., $\Omega_\mathrm{f} = \emptyset$, the SMT decoding will iteratively eliminate infeasible implementations until the pseudo-Boolean solver returns a contradiction. At this moment, SMT decoding has proven that no feasible implementation exists which is neither possible for DSE approaches that solely rely on metaheuristic optimization nor for exact approaches that may only consider linear constraints.

7.4.3 Learning Schemes

For SMT decoding, three learning schemes have been proposed in literature: *Simple learning* requires no problem-specific knowledge while *early learning* requires that the specification allows to also derive *partial implementations*; see [24]. A third scheme that relies on the *deduction of justifications* requires that enough problem-specific knowledge is available such that the background theory can basically derive the *reason* why an implementation is infeasible; see [26]. In the following subsection, all three schemes are introduced.

7.4.3.1 Simple Learning

The simple learning scheme has already implicitly been mentioned in the introduction of SMT decoding. It is a direct implementation of the SMT decoding idea: The solver derives a variable assignment $\mathbf{x}$ which is passed to the background theories. If any of these recognizes that the respective implementation is infeasible, the variable assignment is added to the set of infeasible implementations $\overline{\Psi_N}$, i.e., $\overline{\Psi_N} := \overline{\Psi_N} \vee \mathbf{x}$.

Simple learning is depicted in Fig. 7.13: The triangle shall visualize the decision tree of the solver which is given by the Boolean variables and their phases. One path in that tree is one concrete variable assignment $\mathbf{x}$ and the solver will, of course, only consider those variable assignments that are feasible with respect to the set of linear

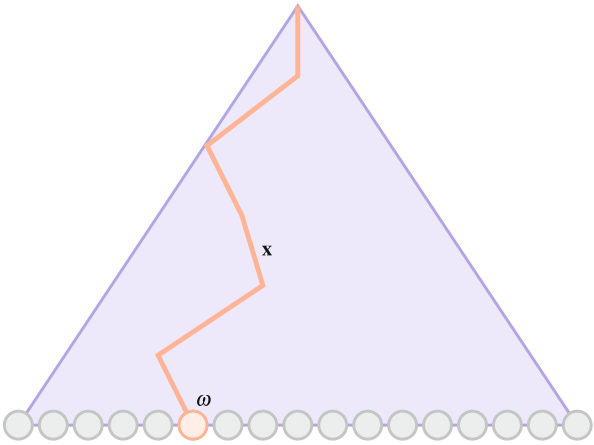


Fig. 7.13 The decision tree that is given from the encoding Ψ_ς. The leaves of the tree denote the set of implementations Ω_L that are feasible with respect to a given set of linear constraints. In the simple learning scheme, each complete variable assignment $\mathbf{x}$ or implementation ω, respectively, is checked for feasibility by the background theories. Thus, the simple learning scheme can only eliminate individual implementations in case of infeasibility

constraints. The leaves of the tree are all variable assignments or points in our search space that are feasible with respect to the set of linear constraints. As can be seen, simple learning considers each point in the search space individually and checks its feasibility; possibly forbidding it for future solving by means of learning.

The advantage of the simple learning is that the background theories can be treated as black-box analysis approaches. The simple learning delivers complete implementations, asks for feasibility using the background theories, and – if required – eliminates a complete implementation from the search space. The drawback is that, for large search spaces, many very similar implementations might exist that all violate certain non-linear constraints. With simple learning, those would have to be checked individually which might be computationally expensive. In the worst case, no feasible implementation might exist such that simple learning becomes an exhaustive search. Note that not only the sheer number of checked implementations might become a problem, but also the huge number of PB constraints that are iteratively added to $\overline{\Psi_N}$ which may become a problem for efficiently solving of the resulting function Ψ_f.

7.4.3.2 Early Learning

The early learning scheme tries to overcome the outlined problems of the simple learning scheme by trying to evaluate already *partial implementations* [24]. The idea of early learning is depicted in Fig. 7.14. There, already a *partial variable assignment* $\mathbf{x}'$ which corresponds to a partial implementation ω' is checked by the background theories for feasibility. The significant advantage arises in case such a partial variable assignment is infeasible: Not only one implementation, but all complete implementations that are based on the partial implementation can be eliminated from the search space at once by $\overline{\Psi_N} := \overline{\Psi_N} \vee \mathbf{x}'$.

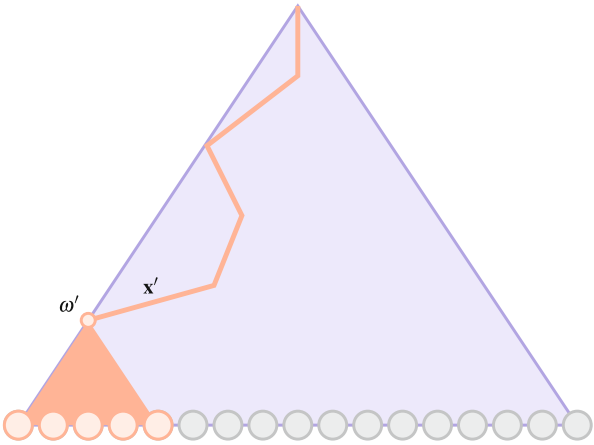


Fig. 7.14 Depicted is again the decision tree given by the encoding Ψ_S. In the early learning scheme, already a partial variable assignment $\mathbf{x}'$ that represents a partial implementation ω' is checked for feasibility by the background theories. Given the partial implementation is already infeasible, a complete subtree of the decision tree can be eliminated, eliminating several – in the concrete example five – implementations at once

However, opposed to the simple learning scheme, care must be taken when applying the early learning scheme.

For early learning, it has to hold that in case a partial implementation is infeasible, all implementations that contain this partial implementation must be infeasible as well. This holds true if the background theory is *monotonic* with respect to partial implementations.

Of course, whether this assumption holds or not heavily depends on the used background theory: Consider, for example, our previous constraint on timeliness of an implementation and a schedule analysis used as background theory. If already a partial implementation violates a given deadline, it will typically hold that a complete implementation with more workload and/or interference in the system will also violate the deadline. For many timing analysis approaches, early learning can be used. On the other hand, a consideration of system reliability may result in a different situation. While a partial implementation might not satisfy a constraint on minimal lifetime, a complete implementation might add additional redundant resources or tasks to the system. With this redundancy, the lifetime criterion might again be met by the complete implementation. But, as discussed in [24], a clever and problem-specific variable ordering might allow to employ early learning at *safe points* in the decision tree such that monotonicity of the background theory is achieved.

7.4.3.3 Deducing Justifications

We recognized that the early learning scheme already offers several advantages over the simple learning strategy but requires a monotonic background theory by deriving partial implementations at *safe points* during the run of the solver. This could be avoided by the simple learning scheme which, however, comes at the drawback of only being able to eliminate one implementation per check. A third learning scheme that explicitly targets this problem is based on the following idea:

> The violation of a certain non-linear constraint is typically not caused by *all* assigned decision variables, but only by a subset of critical decisions termed *justification*.

Consider again the example of a background theory that analyzes timeliness. The violation of a deadline is typically caused by the critical path in the implementation. However, not all design decisions contribute to the critical path, but only a subset of design decisions that cause interference on computation and communication resources. The key idea of this learning scheme is to rely on background theories that consider an implementation ω and the respective variable assignment $\mathbf{x}$, check its feasibility, and deliver the justification $\widetilde{\mathbf{x}}$. Eliminating the justification from the search space has the immediate effect that not only one – as in early learning – but multiple complete subtrees can be removed from the decision tree or search space, respectively. In particular, it eliminates all implementations that contain the determined justification or, in other words, that include the critical decisions that will always result in a violation of the constraint. The concept of this learning scheme is depicted in Fig. 7.15.

Similar to the early learning scheme, it has to hold that all implementations that contain the justification do violate the respective constraint. However, opposed to ensuring this via a respective variable ordering and interrupting the solver, this learning scheme only relies on the simple learning considering the solver while it is the task of the background theory alone to determine the justification. Thus, it can be concluded that the deduction of justifications can be considered the least invasive and most flexible learning approach, given a respective background theory is available.

7.5 Applications

The introduced techniques of SAT and SMT decoding may be employed to a variety of constrained combinatorial problems, of which several are highly relevant for hardware/software codesign at system level. In this section, we will briefly outline some concrete applications of the introduced techniques to serve as directions for further reading and to give evidence of the flexibility and applicability of the underlying ideas.

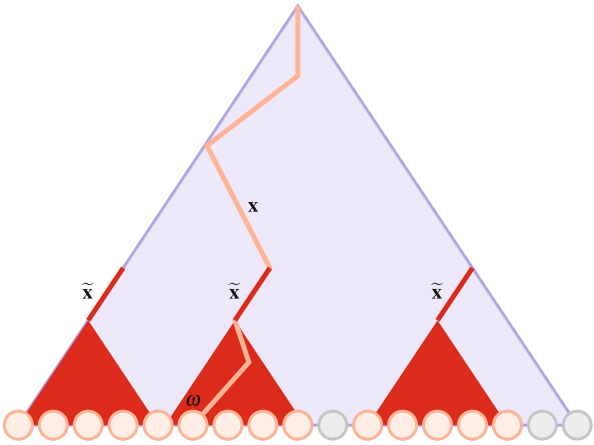


Fig. 7.15 Depicted is again the decision tree given by the encoding Ψ_ς. As in the simple learning scheme, a variable assignment $\mathbf{x}$ that represents an implementation ω is checked for feasibility by the background theories. But here, the background theory has the capability to derive the set of variables and their phase $\widetilde{\mathbf{x}}$ called justification that causes the violation of non-linear constraints. Learning this justification, not only one, but possibly multiple complete subtrees of the decision tree can be eliminated at once

SAT decoding is successfully applied to system-level synthesis problems from the area of Multi-Processor System-on-Chip (MPSoC) design; see, for example, [21]. There, the focus is on the distribution of process tasks to multiple processing units as well as hardware accelerators and also to find a specification encoding that suits the application's Model of Computation (MoC); see ▸ Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign". For upcoming many-core architectures that often feature regular communication topologies such as meshes, SAT decoding is extended to mitigate the complexity increase of the routing in such architectures; see [10]. For many-core architectures and so-called hybrid mapping approaches (see ▸ Chap. 10, "Design Space Exploration and Run-Time Adaptation for Multicore Resource Management Under Performance and Power Constraints"), SAT decoding is used as part of the design-time DSE [31].

A particular domain where the concepts of SAT and SMT decoding are applied is networked embedded systems as can be found in avionics, rail, industrial automation, and automotive systems. Here, routing data over multiple different field bus systems is one problem where SAT decoding enables a conclusive solution [17], particularly for automotive Electric and Electronic (E/E) architectures. Besides the integration of various applications, SAT decoding is also used to integrate additional features such as diagnosis applications [25] like Built-In Self-Tests (BISTs) that must not interfere with the applications yet enhance the quality of the system. SMT decoding is applied to automotive applications with stringent real-time requirements in [26]. The approach in [14] uses a concept similar to SMT decoding to design automotive systems that are completely time-triggered and combines architectural and timing optimization in a unified DSE.

SAT decoding has also been used for the design of dependable embedded systems where dependability-enhancing techniques such as the binding of redundant process or communication tasks are integrated directly into the specification encoding; see, for example, [7]. The application of SMT decoding to consider dependability constraints such as a minimal expected lifetime is discussed in [24].

Finally, modern embedded systems may not only implement a fixed set of applications but rather enable customers to select various features and, thus, create their individual *variant* of the system. Particularly in the automotive domain, *variant management* requires to keep track of both the variants arising from a combination of different applications and the underlying architecture that has to support the different application variants in an efficient fashion. The approaches in [9] and [8] target these problems using the SAT-decoding technique.

Availability of the techniques: The described SAT-decoding approach is publicly available at [18] as part of the open-source library OPT4J [19] which can serve as a base for the application of SAT decoding to a wide range of constraint combinatorial problems. An open-source library termed OPENDSE is also publicly available [20] which already combines the SAT-decoding engine of OPT4J with a system model suitable for system-level DSE and hardware/software codesign as introduced in this chapter.

7.6 Conclusion

This chapter introduces a hybrid optimization approach to be used during Design Space Exploration (DSE) for system-level hardware/software codesign. The targeted problem is that linear as well as non-linear constraints may render many system implementations infeasible, such that classic DSE approaches can hardly find high-quality implementations or – in extreme cases – cannot even find a single feasible system implementation. The main focus of this chapter is the introduction of a hybrid optimization approach that allows a metaheuristic optimization to derive feasible implementations using a nested exact technique. The first method termed SAT decoding is capable of considering linear constraints and is used to introduce the general concept of *hybrid optimization*. Since hardware/software codesign at system level typically also has to respect constraints that cannot be expressed as linear constraints such as on timeliness, power consumption, or reliability, a second approach is introduced that may also handle additional non-linear constraints. This approach termed SMT decoding is capable of employing any available analysis technique to judge whether an implementation violates a given set of non-linear constraints or not and, thus, learns which implementations are infeasible in an iterative but efficient way. Moreover, three different learning schemes are introduced that either require no problem-specific knowledge at all or can significantly improve the learning via the evaluation of partial implementations or the deduction of the cause of a constraint violation. The chapter is concluded with examples of the successful application of the SAT and SMT decoding approaches to different areas such as MPSoC design and automotive systems.

References

1. Blickle T, Teich J, Thiele L (1998) System-level synthesis using evolutionary algorithms. Des Autom Embed Syst 3(1):23–58
2. Coello Coello CA (2002) Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. Comput Methods Appl Mech Eng 191(11–12):1245–1287
3. Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Commun ACM 5(7):394–397
4. De Moura L, Bjørner N (2011) Satisfiability modulo theories: introduction and applications. Commun ACM 54(9):69–77
5. Gajski DD, Kuhn RH (1983) New VLSI tools. IEEE Comput 16(12):11–14
6. Gerstlauer A, Haubelt C, Pimentel A, Stefanov T, Gajski D, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput Aided Des Integr Circuits Syst 28(10):1517–1530
7. Glaß M, Lukasiewycz M, Reimann F, Haubelt C, Teich J (2010) Symbolic system level reliability analysis. In: Proceedings of the international conference on computer-aided design (ICCAD), San Jose, pp 185–189
8. Graf S, Glaß M, Teich J, Lauer C (2014) Multi-variant-based design space exploration for automotive embedded systems. In: Proceedings of design, automation and test in Europe (DATE), p 6
9. Graf S, Glaß M, Wintermann D, Teich J, Lauer C (2013) IVaM: implicit variant modeling and management for automotive embedded systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), p 10
10. Graf S, Reimann F, Glaß M, Teich J (2014) Towards scalable symbolic routing for multi-objective networked embedded system design and optimization. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 2:1–2:10
11. Hernandez-Aguirre A, Botello-Rionda S, Coello Coello CA, Lizarraga-Lizarraga G, Mezura-Montes E (2004) Handling constraints using multiobjective optimization concepts. Int J Numer Methods Eng 59(15):1989–2017
12. Kienhuis ACJ (1999) Design space exploration of stream-based dataflow architectures – methods and tools. Ph.D. thesis, Delft University of Technology
13. Le Berre D, Parrain A (2010) The Sat4J library, release 2.2. system description. J Satisf Boolean Model Comput 7:59–64
14. Lukasiewycz M, Chakraborty S (2012) Concurrent architecture and schedule optimization of time-triggered automotive systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 383–392
15. Lukasiewycz M, Glaß M, Haubelt C, Teich J (2007) Solving multiobjective Pseudo-Boolean problems. In: Proceedings of the international conference on theory and applications of satisfiability testing (SAT), pp 56–69
16. Lukasiewycz M, Glaß M, Haubelt C, Teich J (2008) Efficient symbolic multi–objective design space exploration. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC), Seoul, pp 691–696
17. Lukasiewycz M, Glaß M, Haubelt C, Teich J, Regler R, Lang B (2008) Concurrent topology and routing optimization in automotive network integration. In: Proceedings of the design automation conference (DAC), Anaheim, pp 626–629
18. Lukasiewycz M, Glaß M, Reimann F Opt4J–meta-heuristic optimization framework for java. http://www.opt4j.org/
19. Lukasiewycz M, Glaß M, Reimann F, Teich J (2011) Opt4J: a modular framework for meta-heuristic optimization. In: Proceedings of the genetic and evolutionary computation conference (GECCO), pp 1723–1730
20. Lukasiewycz M, Reimann F OpenDSE–open design space exploration framework. http://opendse.sourceforge.net/

21. Lukasiewycz M, Streubühr M, Glaß M, Haubelt C, Teich J (2009) Combined system synthesis and communication architecture exploration for MPSoCs. In: Proceedings of design, automation and test in Europe (DATE), pp 472–477
22. Prakash S, Parker AC (1992) SOS: synthesis of application-specific heterogeneous multiprocessor systems. J Parallel Distrib Comput 16(4):338–351
23. Puchinger J, Raidl G (2005) Combining metaheuristics and exact algorithms in combinatorial optimization: a survey and classification. In: Proceedings of the first international work-conference on the interplay between natural and artificial computation (IWINAC), vol 3562, pp 41–53
24. Reimann F, Glaß M, Haubelt C, Eberl M, Teich J (2010) Improving platform-based system synthesis by satisfiability modulo theories solving. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 135–144
25. Reimann F, Glaß M, Teich J, Cook A, Gómez LR, Ull D, Wunderlich HJ, Abelein U, Engelke P (2014) Advanced diagnosis: SBST and BIST integration in automotive E/E architectures. In: Proceedings of the design automation conference (DAC), p 8
26. Reimann F, Lukasiewycz M, Glaß M, Haubelt C, Teich J (2011) Symbolic system synthesis in the presence of stringent real-time constraints. In: Proceedings of the design automation conference (DAC), pp 393–398
27. Smith AE, Coit DW (1997) Penalty functions, chap. C 5.2. Institute of Physics Publishing and Oxford University Press, Bristol
28. Teich J (2012) Hardware/software co-design: past, present, and predicting the future. Proc IEEE 100(5):1411–1430
29. Teich J, Blickle T, Thiele L (1997) An evolutionary approach to system-level synthesis. In: Proceedings of the international workshop on hardware/software codesign (CODES/CASHE), pp 167–171
30. Teich J, Haubelt C (2007) Digitale hardware/software-systeme: synthese und optimierung, 2nd edn. Springer, Heidelberg
31. Weichslgartner A, Gangadharan D, Wildermann S, Glaß M, Teich J (2014) DAARM: design-time application analysis and run-time mapping for predictable execution in many-core systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 34:1–34:10
32. Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. IEEE Trans Evol Comput 3(4):257–271

Architecture and Cross-Layer Design Space Exploration

8

Santanu Sarma and Nikil Dutt

Abstract

The task of architectural Design Space Exploration (DSE) is extremely complex, with multiple architectural parameters to be tuned and optimized, resulting in a huge design space that needs to be explored efficiently. Furthermore, each architectural parameter and/or design point is critically affected by decisions made at lower levels of abstraction (e.g., layout, choice of transistors, etc.). Ideally designers would like to perform DSE incorporating information and decisions made across multiple layers of design abstraction so that the ensuing design space is both feasible and has good fidelity. Simulation-based methods alone can not deal with this incredibly large and complex design space. To address these issues, this chapter presents an approach for cross-layer architectural DSE that efficiently prunes the large design space and furthermore uses predictive models to avoid expensive simulations. The chapter uses a single-chip heterogeneous single-ISA multiprocessor as an exemplar to demonstrate how the large search space can be covered and evaluated efficiently. A cross-layer approach is presented to cope with the complexity by restricting the search/design space through the use of cross-layer prediction models to avoid too costly full system simulations, coupled with systematic pruning of the design space to enable good coverage of the design space in an efficient manner.

Acronyms

CLDSE	Cross-Layer Design Space Exploration
DoE	Design of Experiments

S. Sarma (✉)
University of California Irvine, Irvine, CA, USA
e-mail: santanus@uci.edu

N. Dutt
Center for Embedded and Cyber-Physical Systems, University of California Irvine, Irvine, CA, USA
e-mail: dutt@ics.uci.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_9

DSE	Design Space Exploration
EDP	Energy-Delay Product
EDSP	Energy-Delay Square Product
HMP	Heterogeneous Multi-core Processor
ILP	Instruction-Level Parallelism
ISA	Instruction-Set Architecture
RSM	Response Surface Modeling
SA	Simulated Annealing

Contents

8.1 Introduction

The task of architectural Design Space Exploration (DSE) is extremely complex, with multiple architectural parameters to be tuned and optimized, resulting in a huge design space. Furthermore, each architecture parameter and/or design point is critically affected by decisions made at lower levels of abstraction (e.g., layout, choice of transistors, etc.). Ideally, designers would like to perform DSE incorporating information and decisions made across multiple layers of design abstraction so that the ensuing design space is both feasible and has good fidelity. Simulation-based methods alone can not deal with this incredibly large and complex design space. To address these issues, this chapter presents an approach for cross-layer architectural DSE that efficiently prunes the large design space, and furthermore uses predictive models to avoid expensive simulations. The chapter uses a single-chip heterogeneous single-ISA multiprocessor system as an exemplar to demonstrate how the large search space can be covered and evaluated efficiently. This chapter complements other chapters in this book that give additional insights on specific optimization and exploration strategies. For instance: Chapter 6 by Scuito et al. describes optimization strategies for DSE; Chapter 7 by Glass et al. details advanced hybrid DSE techniques; and Chapter 10 by Henkel et al. incorporates power-aware run-time adaptions in DSE.

Single-chip-single-ISA-based Heterogeneous Multi-core Processors (HMPs) are increasingly considered as an attractive design alternative to homogeneous

multiprocessor systems because of their superior performance, power, and energy efficiency while providing the flexibility of using the same software (binaries) and development tools across cores for a range of applications. HMPs can effectively address complex requirements of diverse applications by executing workloads (or tasks) in the most appropriate core types to meet competing and conflicting objectives and figures of merit (e.g., performance, power, energy, throughput, area, cost etc.) [3, 19, 29, 30]. Since different workloads (e.g., CPU bound, integer-intensive, floating-point intensive, memory intensive, etc.) require different resources, a key issue is to determine and select the right types and number of cores (processing elements) for an allocation strategy that maps the workload (or tasks) to right core type such that the type of workload will best benefit from the given platform. The selection of number and type of cores is not straightforward when the applications executed by these HMPs exhibit diverse workload characteristics. When designing such a system, a chip architect must decide how to distribute the available limited system resources, such as area and power, among all the processor cores.

HMPs that integrate a mix of small power-efficient cores and big high-performance cores are attractive alternatives to homogeneous multiprocessor systems because they have the potential for higher performance and reduced power consumption. Contemporary mobile phones have already embraced hardware core heterogeneity, for instance, ARM's big.LITTLE architecture [17] and NVIDIA's Kal-El [39] that have cores of different strengths in one cache coherence domain with the same Instruction-Set Architecture (ISA). Architectures with more than two core types are already a reality (e.g., NVIDIA's Kal-El [39] that integrates four high performance cores, one low performance core, and many GPU cores), and this trend toward heterogeneity is only expected to grow further in the future. Processor cores in a heterogeneous multi-core system can differ in their static microarchitectures to dynamic behavior or modes of operation (e.g., frequency or operating voltage). Broadly, the vast space of HMPs can be classified by core type (strength/size/number/ISA) and heterogeneity levels. As an example, consider Fig. 8.1 that shows a sample space of HMP architectures using a combination of different ARM cores – ranging from big (A15) to medium (A11) to small (A7) – that vary in their performance, power, and energy efficiency.

HMPs provide architecturally diverse cores with drastically different power-performance trade-offs that can be exploited for system efficiency. Consequently, HMP architectures and their DSE is an active area of research. DSE is the process of discovering and evaluating design alternatives during system development that enable design optimization, system integration, and rapid prototyping prior to implementation. The main challenge of DSE for HMPs arises from the sheer size of the design space that must be explored because a typical HMP system has a huge number of possibilities (in the millions, if not billions), and so enumerating every point in the design space considering different layers of the system stack is prohibitive. Although several works have studied HMPs and their run-time systems [2, 4, 8, 24, 29, 36, 47], the topic of DSE of HMPs is still in its infancy and needs a principled approach as these architectures evolve in their diversity and complexity.

DSE of HMPs is critical to evaluate and architect a suitable processor platform configuration. Selection and composition of the platform is important at the early

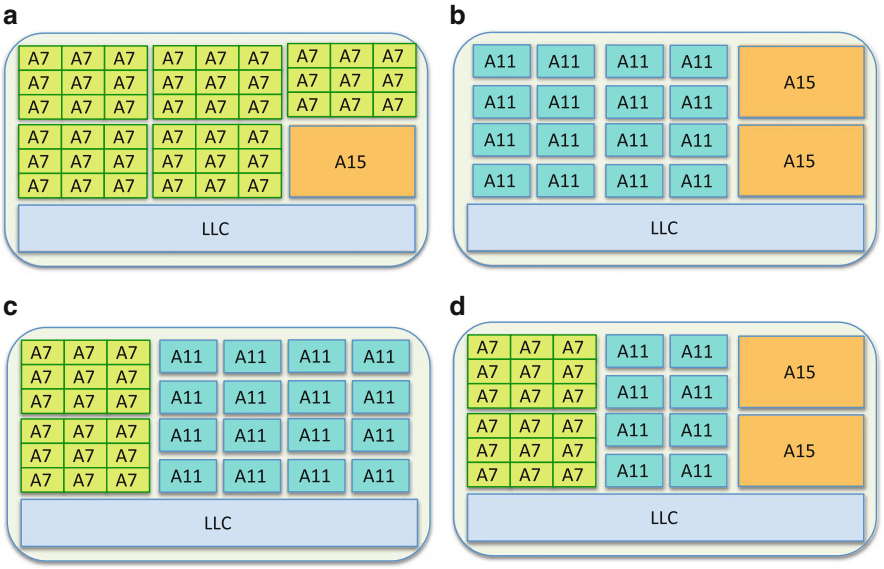


Fig. 8.1 Examples of heterogeneous architectures composition for the same die area using big (A15), medium (A11) , and little (A7) cores

stage of the design, and a chip architect must decide how to distribute the available limited system resources, such as area and power, among all the processor cores. Moreover, since HMPs are inherently designed as a multilayered system, either gross approximation or complete neglect of any layer and its features can affect the behavior, misrepresent the intricate multilayer trade-offs and interactions, as well as misguide the design and composition process. However, due to their diverse and vast design space, selecting a suitable HMP configuration with different core types within a given area-power budget is an extremely challenging task. This complexity challenge can be overcome in an intelligent manner by (a) restricting the search/design space and (b) using cross-layer prediction models to selectively avoid too costly full system simulations all the time, making a good coverage of the design space affordable. The problem of exploring and configuring an HMP for a given system goal under system-level constraints (such as equal area or power budget) can be cast as a cross-layer optimization problem. This chapter illustrates an approach that jointly consider cross-layer features of the application, operating system (task allocation strategies), and hardware architecture while deploying computationally efficient predictive models (of performance and power) in configuring the HMP platform resources (number and types of cores) in an evolutionary optimization framework. The predictive cross-layer approach enables the designer to comparatively evaluate and select the most promising (e.g., energy and performance efficient) HMP configuration in over two order of magnitude less simulation time compared to a full system simulation especially during the early design and verification stages when the design space is at its largest.

8.2 Design Space Exploration of Heterogeneous Multi-core Processors

Design space exploration needs two important components (a) a simulation infrastructure to evaluate different configurations and (b) predictive models to assess the quality of new configurations with a metric of goodness. System designers often build performance, power, and area (PPA) models of a microarchitecture to predict these metrics as a function of the design parameters $\mathbf{x}$. These models may be often represented as $y = J(\mathbf{x})$, where $J(\mathbf{x})$ represents a cycle accurate simulator or empirical model fit to simulated [33, 51] or prototype system data [10]. Detailed simulation is the most widely adopted approach in evaluating $J(\mathbf{x})$. However, the significant computational costs of simulation often hinder the design process leading to approaches that aim at reducing the simulation cost by either reducing the number of simulation runs of an evaluated program [14,22,41,51] or reducing the number of simulated architectures by building predictive models [7,9,11,20,23,31,33,40,50]. A combination of jointly reducing both program and architecture simulations is also possible [13,27].

Design space exploration of HMPs is much more complex and challenging than that of homogeneous architectures since it involves reevaluating architecture and application options along with the operating system (OS) implications [10]. A straightforward extension of the abovementioned works is not directly applicable for HMPs as they are targeted toward homogeneous multi-core processors without considering the operating system. The exploration is performed using architectural and program space parameters without considering the OS scheduling by either using microarchitecture or cycle-accurate simulation. In order to consider the operating system in the DSE, full system cycle-accurate architectural simulators are indispensable tools for evaluating complicated and subtle design trade-offs with respect to large design spaces and handling various design constraints. As an exemplar, this chapter presents an extended full system cycle accurate HMP simulator [46] based on Gem5 [6] along with the Linux OS as the infrastructure for the cross-layer DSE. A predictive model of the full system is built, considering parameters from all the system stacks in order to capture the HMP performance and power characteristics. Unlike the state-of-the-art mentioned earlier that focused either on uni-core or homogeneous multi-core processors, the predictive approach specifically focuses on the cross-layer predictive-model-based DSE of the heterogeneous multi-core processor while considering key parameters of the application, architecture, and the operating system in the evaluation of the HMP configurations.

This chapter specifically presents a cross-layer approach for exploring and configuring a HMP for a given goal under system-level constraints (such as equal area or power budget) based on recent work [45]. Unlike the state-of-the-art approaches, the presented approach jointly considers features of the application, operating system (task allocation strategies), and hardware architecture while deploying computationally efficient predictive models (of performance and power) in composing the HMP platform resources (number and types of cores). The predictive cross-layer approach enables the designer to comparatively evaluate and select the

most promising (e.g., energy and performance efficient) HMP configuration in over two order of magnitude less simulation time especially during the early design and verification stages when the design space is at its largest.

8.2.1 Design of Experiments

The term "experiment" [28, 37] concerns situations where we have to organize a systematic scientific procedure to obtain some meaningful information about an object of interest. Design of Experiments (DoE) [1, 43] is an efficient and scientific approach that considers all factors simultaneously, applied to an experimentation to obtain meaningful information and to determine the relationship between factors affecting a process and the output of that process. DoE is a powerful tool that can be used in a variety of experimental platforms where more than one input factor is suspected of influencing an output. DoE allows for multiple input factors to be manipulated determining their effect on a desired output. By manipulating multiple inputs at the same time, DoE can identify important *interactions* that may be missed when experimenting with one factor at a time. Likewise, DoE provides a full insight of interaction between design elements; therefore, it helps turn any ad hoc design process into a robust, predictable process.

The DoE methodology may involve *controllable* and *uncontrollable* input factors. *Controllable* input factors are those input parameters that can be modified in an experiment while *uncontrollable* input factors cannot be changed. These factors need to be recognized to understand how they may affect the output or *response*. DoE provides information about the interaction of these factors and the total system work flow, something not obtainable through testing one factor at a time while maintaining other factors constant. Additionally, DoE shows how interconnected factors respond over a wide range of values, without requiring the testing of all possible values directly. Often, hypothesis testing is performed to determine the significant factors using statistical methods in combination with *orthogonal* vectors or sets of *orthogonal* vectors that are uncorrelated and independent [1]. Before doing the actual experiment, experimental design requires careful consideration of several factors such as number of factors that influence the design, fixed or random levels of these factors, control conditions required in the design process, sample size, number of units collected for the experiment to be generalizable, the relevance of interactions between factors, noise, etc., for the establishment of validity, reliability, and replicability [1]. In order to predict the output responses for any given combination of input values, DoE fits response data to mathematical models or predictive models. With these models, it is possible to optimize critical responses and find the best combination of input values. As measurements are usually subject to variation and measurement uncertainty, *blocking* and *replication* of experiments are adopted to overcome these shortcomings. Blocking is the arrangement of experimental units into groups consisting of units that are similar to one another in order to avoid any unwanted variations in the input or experimental process and thus allows greater precision in the estimation of the source of variation under study. A *randomization* process is used to assign individuals at random

to groups or to different groups in an experiment so that each individual of the population has the same chance of becoming a part in the process. Similarly, replication of the experiments (i.e., perform the same combination run more than once) is done in order to identify the sources of variation, to get an estimate for the amount of random error that could be part of the process, and to further strengthen the experiment's reliability and validity.

8.2.2 Response Surface Models

Response Surface Modeling (RSM) techniques allow determining an analytical relationship or dependence between several design parameters and one or more response variables into a mathematical framework typically to rapidly evaluate a system-level metric. The working principle of RSM exploits a set of simulations generated by DoE in order to obtain a predictive model that is also called a response model. A typical RSM flow involves a training phase in which known data (or training set) is used to identify the RSM configuration and a prediction phase in which the RSM is used to forecast or predict unknown system response. These predictive models can be developed using established techniques [40] such as interpolations, linear regression, or artificial neural networks. RSM methodology has been used extensively to study the design space exploration of homogeneous architectures [35, 40]. The next section outlines the methodology specifically adapted for emerging HMP architectures.

8.3 Cross-Layer Predictive Model Building Approach

This section presents a Cross-Layer Design Space Exploration (CLDSE) approach, a methodology that allows evaluation of large architectural design spaces at different levels of abstraction to achieve efficiency (e.g., reducing simulation time by trimming down the large design space into a small finite set of points) and accuracy (gradual refinement of the abstraction models). The cross-layer-based DSE for the HMPs is motivated by the platform-based approach [26,42] with the difference that the hardware architecture platform and the mapping strategy are varied along with diverse spectrum of applications for given system-level constraints. The presented methodology combines the DoE [43] and predictive model [40] techniques to predict the quality of the nonsimulated design points thereby speeding up the exploration process while reducing the number of required simulations. While the DoE phase generates an initial plan of experiments used to create a coarse view of the target design space to be explored by simulations, the predictive model – a closed-form expression of objective (figure of merit) space as a function of the parameter space – is useful during the DSE phase to quickly converge to the Pareto set of the multi-objective problem without executing lengthy simulations. The modeling and optimization techniques proposed in [33, 40] are used to iteratively update the predictive models (as shown in Fig. 8.2) while simulating different parts of the system stack.

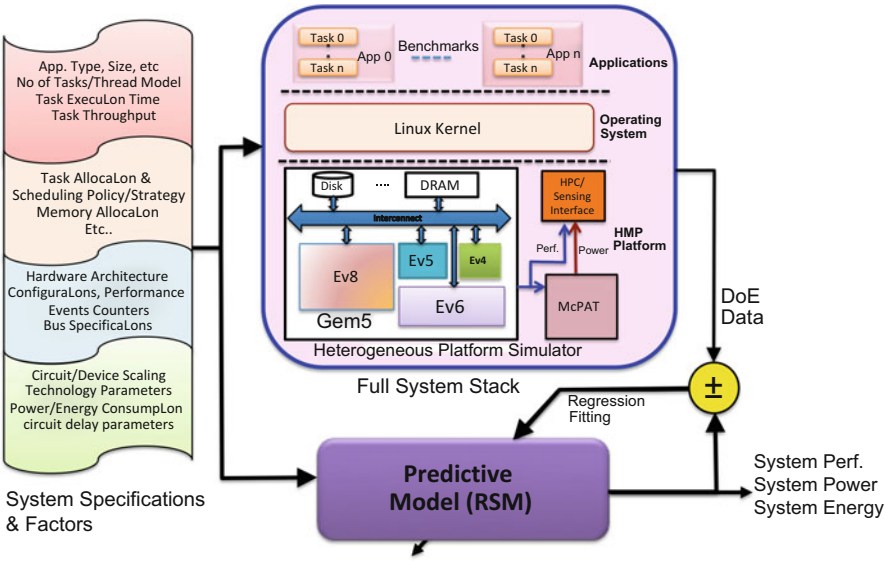


Fig. 8.2 Training phase: cross-layer predictive model building approach

The cross-layer predictive modeling methodology for HMP architectures is divided in two phases. In the training phase, known data (from a training set) are used to identify the predictive model configuration as depicted in Fig. 8.2. A special set of benchmarks are used for coverage of the design space. On the other hand, during the prediction phase, a predictive model is used to forecast the unknown system response as illustrated in Fig. 8.3. The training phase of the cross-layer predictive modeling approach captures the architectural design spaces and behaviors at different levels of abstraction to achieve efficiency (e.g., reducing simulation time by trimming down the large design space into a small finite set of points) and accuracy (gradual refinement of the abstraction models). This approach, illustrated in Fig. 8.2, is motivated by the platform-based approach [26,42] with the difference that the hardware architecture platform and the mapping strategy are varied along with diverse spectrum of applications for given system-level constraints. The modeling and optimization techniques proposed in [40] are deployed to iteratively update the predictive models (as shown in Fig. 8.2) of different parts of the system stack as discussed in the subsequent sections.

8.3.1 Problem Formulation

Consider a shared memory HMP architecture as shown in Fig. 8.1 consisting of K types of core represented using a set $\Pi = \{\pi_1, \pi_2, \ldots, \pi_K\}, \pi_i \neq \pi_j, K > 1$ having

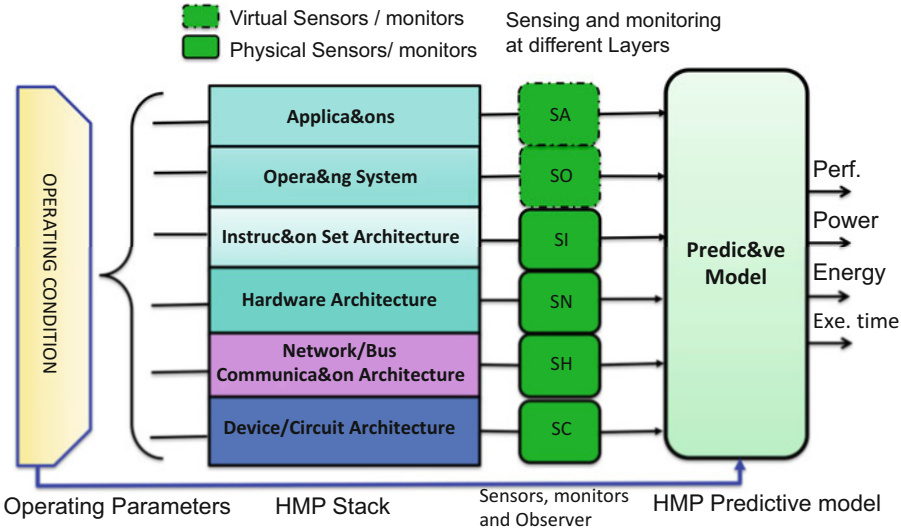


Fig. 8.3 Prediction phase: use of the cross-layer predictive model using features from different layers of the stack during prediction

corresponding areas $\Lambda = \{a_1, a_2, \ldots, a_K\}$. Let the set of all processing elements be $PE = \{p_1, p_2, \ldots., p_n\}$ where p_j is an instance of a core type in Π. The HMP consists of the core combinations $\mathbf{C} = \{n_1, n_2, .., n_K\}$ such that the total number of processors in the HMP is $n = \sum_{l=1}^{K} n_l$, where n_l is the number of processors of type π_l and the total area $A = \sum_{i=1}^{K} a_i * n_i$. Let $A_{\max}$, $P_{\max}$ be the respective maximum die area and allowable power consumption in the design of the HMP. The goal is to find the HMP core combinations $\mathbf{C}$ such that a platform objective $\mathbf{J}$ (e.g., power/energy efficiency) is optimized under system constraints (such as area or power) as below:

$$\begin{aligned} &\text{Maximize} \quad (\mathbf{J}) \quad . \\ &\mathbf{C} \\ &s.t \quad A \leq A_{\max} \end{aligned} \tag{8.1}$$

As the design space for HMP composition problem in (8.1) is extremely large, a few assumptions and approximations are made to reduce the design space. First, assume that the number of core types K is small (<5). Second, as there is a hard constraint on system area and power resources, the composition space of $\mathbf{C}$ can be reduced to a set of feasible configurations $\mathbb{C} = \{C_1, C_2, .., C_N\}$. Then the problem is to chose one of these C_i for a given set of workloads and OS level workload allocation strategy that optimizes the system goal $\mathbf{J}$.

8.3.2 Application and Workload Models

The workload and their diversity (program phases , CPU, memory, and IO intensive workloads) are modeled using a task-based (thread-based) model and are exposed to the OS using the performance and power/energy characterization matrices where:

- $V = \{v_i\}$ is the set of tasks (or interchangeably used for threads). v_i stands for task i where $1 \leq i \leq m$ and m is the number of tasks. Without loss of generality, a task or group of tasks represents the workload/application. Let N_i represent the computational workload (measured in terms of number of instructions) of task v_i.
- $\mathbf{S} = [\mathbf{ips_i}] = \left\{\text{ips}_{ij}, 1 \leq i \leq m, 1 < j \leq n\right\}$ is the average throughput matrix (measured in terms of instructions per second) when executing the tasks on different processors. $\text{ips}_{ij} (= \text{IPC}_{ij} * F_j)$ represents the average throughput when task v_i executes on processor p_j and is the product of the IPC_{ij} (instruction per cycle) and the frequency of the core F_j. The IPC_{ij} can be measured directly from the processor's built-in performance counters [15].
- $\Gamma = \left[N_i / s_{ij}\right] = \left\{\tau_{ij}, 1 \leq i \leq m, 1 < j \leq n\right\}$ is the average execution time (or the time span) matrix. τ_{ij} is the average execution time of task v_i on processor p_j.
- $\mathbf{P} = [\mathbf{pw_i}] = \left\{pw_{ij}, 1 \leq i \leq m, 1 < j \leq n\right\}$ is the average power consumption matrix of tasks executing on different processors. $\mathbf{pw}_i = \{pw_{ij}, 1 \leq j \leq n\}$ represents a vector of all the average powers of task v_i executing on each processor. pw_{ij} represents the average power of task v_i executing on processor p_j, and it varies with time. The power consumption pw_{ij} of a task v_i can be computed by using combination of performance counters [15].
- $\Xi = \left[\varepsilon_{ij}\right] = \left\{pw_{ij} \times \tau_{ij}\right\}$ is the average energy consumption defined as the product of the average power consumption and execution time.

8.3.3 Heterogeneity-Aware Task Allocation

The task allocation problem of multi-core processors within an HMP architecture consists of finding an optimal distribution of tasks on a set of processors $PE = \{p_1, p_2, \ldots, p_n\}$. It is assumed that each processor runs independently but can only run one task at any instant of time. An assignment of all tasks $V = \{v_1, v_2, \ldots, v_m\}$ to available processors $PE = \{p_1, p_2, \ldots, p_n\}$ a "schedule" $\boldsymbol{\Psi}$ is represented as:

$$\begin{gathered} \boldsymbol{\Psi} = \left\{\lambda_j, 1 \leq j \leq n\right\} \\ \lambda_j = \{v_i, 1 \leq i \leq m\}, \forall v_i \in V = \{v_1, v_2, \ldots, v_m\}, \end{gathered} \tag{8.2}$$

where λ_j represents the schedule of set of task for the processor p_j and v_i represents a task among the set of tasks $V = \{v_1, v_2, \ldots, v_m\}$ that is mapped to processor p_j. A schedule as defined in (8.2) will result in a total execution time and power distribution consumption as a function of the task allocation taking into account the

Table 8.1 Heterogeneity-aware task allocation strategies for a given HMP composition

Sl No	Platform design goal	Allocation	Problem definition	Objective function	Nomenclature
1	Performance maximization (PerfMax)	$\min D$	Find Ψ_D $\exists\ J_D$ is minimized	$t_{\text{opt}} = \min\{J_D\} = \min\{\max\{t_j\}\}$ $J_D = \max\{t_j\};\ t_i = \sum_{i=1}^{k} \tau_{ij}$ $1 \leq j \leq n$	t_j represents total execution time of the tasks in processor p_j
2	Energy minimization (EnergyMin)	$\min E$	Find Ψ_E $\exists\ J_E$ is minimized	$\Xi_{\text{opt}} = \min\{J_E\};\ J_E = \sum_{j=1}^{n} \xi_j$ $\xi_j = \sum_{i=1}^{k} \varepsilon_{ij} = \sum_{i=1}^{k} pw_{ij}.\tau_{ij}$; $1 \leq j \leq n$	ξ_j represents sum of total energy consumed by k task in processor p_j
3	Power minimization (PowerMin)	$\min ED$	Find Ψ_{ED} $\exists\ J_{ED}$ is minimized	$J_{ED} = \min\{J_E.J_D\}$	Energy delay product
4	Energy efficiency maximization (EEMax)	$\min ED^2$	Find Ψ_{ED^2} $\exists\ J_{ED^2}$ is minimized	$J_{ED^2} = \min\{J_E.J_D^2\}$	Energy delay square product

heterogeneity of processing elements and workload. In other words, for different allocation strategies, the total execution time and energy consumption in the multicore processor system will be different. Thus, the CLDSE determines a schedule $\boldsymbol{\Psi}$ for the given set of tasks that meets an objective or a performance index as defined in Table 8.1.

8.3.3.1 Optimization Methodology

Finding the optimal allocation that maximizes or minimizes the multiple objective functions is a combinatorial problem; therefore, a solution based on brute force search is not suitable even for a small number of cores and threads due to combinatorial explosion; furthermore, this problem is shown to be NP-hard [49]; thus, polynomial time optimal solutions are not available at all. However, heuristics that exploit specific characteristics of the problem can be adopted to reach acceptable solutions within a reasonable amount of time. Owing to the tremendous diversity of heuristics, a judicious choice of heuristics is critical for finding efficient solutions. Many heuristics often converge to local minima resulting in poor results [12], while others cannot be applied due to the nonlinear nature of the thread allocation objective function (e.g., linear-programming-based approaches [48]). In general, problem structure-dependent heuristics does not provide a generic solution, and a new heuristic formulation has to be obtained for every change in the problem structure (e.g., objective function or constraints).

A more generic approach to such optimization problems have used probabilistic strategies such as *Simulated Annealing (SA)* that have demonstrated the ability to

produce close-to-globally-optimal solutions with a moderate complexity in terms of execution time [12]. SA can easily accommodate changes in the problem nature without significant modifications and provide tunable parameters to trade-off computational complexity for solution quality. Furthermore, SA can also be parallelized and distributed to control the computation complexity for extreme scalability.

8.3.3.2 Simulated Annealing-Based Optimization

Simulated Annealing (SA) is a method for solving unconstrained and bound-constrained optimization problems [12]. The method models the physical process of heating a material and then slowly lowering the temperature to decrease defects, thus minimizing the system energy. At each iteration of the SA algorithm, a new point is randomly generated. The distance of the new point from the current point, or the extent of the search, is based on a probability distribution with a scale proportional to the temperature. The algorithm accepts all new points that lower the objective but also accepts points that raise the objective with a certain probability. By accepting points that raise the objective, the algorithm avoids being trapped in local minima and is able to explore globally for more possible solutions. An annealing schedule is selected to systematically decrease the temperature as the algorithm proceeds. As the temperature decreases, the algorithm reduces the extent of its search to converge to a minimum. The SA-based algorithm outlined in Fig. 8.4 is used as the optimization engine for exploring the cross-layer design space of HMPs.

8.3.4 Predictive Modeling of Performance and Power of Different Core Types

The predictive models based on RSM as described in [40, 43] are closed-form analytical expressions suitable for predicting the quality of nonsimulated design points. Predictive model techniques are typically introduced to decrease the time due to the evaluation of the system-level objective function $\mathbf{J}(\mathbf{x})$ for each architecture $\mathbf{x}$. A response surface model for the function $\mathbf{J}(\mathbf{x})$ is an analytical function $\mathbf{r}(\mathbf{x})$ such that

$$\mathbf{J}(\mathbf{x}) = \mathbf{r}(\mathbf{x}) + \boldsymbol{\epsilon}, \tag{8.3}$$

where ϵ is the estimation error. Typically, an appropriate predictive model for $\mathbf{J}(\mathbf{x})$ is such that it has some desired statistical properties such as a mean of zero and small variance. The working principle of a predictive model is to use a set of simulations generated by DoE in order to build the response model of the system. A typical predictive model-based flow involves a) a training phase, in which known data (or training set) are used to identify the predictive model configuration and b) a prediction phase, in which the predictive model is used to forecast the unknown system response. This chapter demonstrates the use of linear regression techniques to construct the predictive model by taking into account the interaction between the

SA Input Params: Temperature T, Temperature schedule c, Maximum number of iterations $Iter_{max}$
Input Data: HMP config **C**, Throughput Matrix **S**, Power Matrix **P**, Execution Time Matrix Γ, Energy Matrix Ξ
Output: Allocation Ψ

1. Set an initial solution $\Psi = \Psi_0$
2. Obtain a new solution $\Psi' = \Psi$ and randomly perturb one of the elements Ψ'_{ji} of Ψ'. The so-called Bolz-mann generating scheme accomplishes this:

$$idx = j * m + i$$
$$idx = [idx + \sqrt{T} \times rand()] \bmod (n * m)$$
$$j' = idx \bmod n,\ i' = (idx - j')/m$$
$$swap(\Psi'_{ji}, \Psi'_{j'i'})$$

   where $rand()$ generates uniformly distributed random integer numbers.
3. Evaluate the objective function $J(C, S, P, \Gamma, \Xi)$ for Ψ'
4. Accept (set $\Psi = \Psi'$) or reject Ψ'. If the value of the objective function is lower than before the perturbation, always accept. If it is higher, then accept according to the probabilistic rule

$$accept\ if\ rand() < exp\left(\frac{E_0 - E_p}{T}\right)$$

   where $E_0 - E_p$ is the difference in objective function values before and after the perturbation.
5. Decrease the temperature according to the cooling schedule:

$$T = c \times T$$

   where $0 < c < 1$ is a constant.
6. The algorithm stops when the average change in the objective function is small relative to the tolerance, or when it reaches the maximum number of iterations $Iter_{max}$, otherwise, it goes back to step 2.

Fig. 8.4 Heterogeneity-aware static task allocation using SA

parameters and the quadratic behavior with respect to a single parameter using the general model discussed in [40].

In order to concisely encapsulate the effects of performance, power, and workload behavior, an effective approach is required to determine and represent the performance, power, and the energy matrices as described above. A combination of measurement and on-line prediction is used to construct these matrices. Estimation or the prediction of the performance and power matrices are possible as there is a direct correlation between the behavior of different core types. The key idea behind the estimation and prediction of execution time and power matrix relies on the fact that the performance of a task on one core is correlatable to the performance in another core (with the same ISA and memory hierarchy) with a good degree of accuracy. By measuring the performance of the task in one processor, one can predict the performance in other processors. The execution time τ_{ij} of a task v_i on the processor p_j can be defined as,

$$\begin{aligned} \tau_{ij} &= \frac{N_i}{\mathrm{IPC}_{ij} * F_j} = \frac{N_i}{\mathrm{ips}_{ij}} \\ \mathrm{IPC}_{ij} &= 1/CPI_{ij}. \end{aligned} \tag{8.4}$$

Next, core specific performance (throughput) predictors are developed and then combined to obtain performance prediction of the combined total platform. The average throughput IPC_{ij} is for a given task v_i running on processor p_j is predicted by using a linear predictor

$$\text{IPC}_{ij} = \Phi_j * X_{ij}^T, \tag{8.5}$$

where Φ_j is constant vector of a predictive model [2, 24] and $X_{ij}^T = [x_{1i}, x_{2i}, .., x_{qi}]_j^T$ is a characterization vector of core architectural features and hardware counter (cycle counters, instruction counters, performance degradation events) values that is used to predict the performance for the core p_j for the task v_i. The cross-layer features and hardware architecture counters are used in the prediction. The following static features and dynamic hardware performance counters are used:

- **Hardware Architecture Features:** Issue width (I_w), LQ/SQ size (LSQ), IQ size (IQ), ROB size (ROB), Int/float Regs (IFR), L1\$I size (KB) (L_{1I}), L1\$D size (KB) (L_{1D}), Freq. (MHz) (F), voltage (V), core area (a).
- **Performance Events Counters:** The following events are measured that are known to drive the performance of a core [2, 24]: mispredicted branches, which are used to compute the branch misprediction rate (m_B) and instruction/data L1 caches and TLBs misses and hits, which are used to compute the L1 instruction miss rate (m_{L1I}), L1 data cache miss rate (m_{L1D}), instruction TLB miss rate (m_{ITLB}), data TLB miss rate (m_{DTLB}), and Context switch counters (Cw).
- **Cycle and Instruction Counters:** the following cycle counters are sampled: the amount of *busy cycles* (cy_{Busy}), *idle cycles* (cy_{Idle}), and *sleep cycles* (cy_{Sleep}) of a core. *Busy cycles* represent the time a core spends doing computation, *idle cycles* capture idling time due to pipeline stalls or cache misses, and *sleep cycles* capture the time a core spends in a quiescent state. Furthermore, the following instruction counters are sampled: total amount of *committed instructions* (I_{total}), *committed load and stores* (I_{mem}), and *committed branches* (I_{branch}).

Similarly, the power consumption of each task is computed for all the cores by measuring the power consumption in a core and suitably scaling it by the scaling factor among the cores using a linear predictor described below:

$$pw_{ij} = \Theta_j * X_{ij}^T, \tag{8.6}$$

where $\Theta_j = [\theta_1, \theta_2, \ldots, \theta_q]_j$ are constant vectors obtained by fitting the data of the benchmarks and $X_{ij}^T = [x_{1i}, x_{2i}, .., x_{qi}]_j^T$ is the architecture feature and hardware counter (cycle counters, instruction counters, performance degradation events).

The computational complexity (execution time) and accuracy of the predictors for sample benchmarks are shown in Table 8.3.

8.3.5 Training Methodology and Benchmarks

The process of training and the training data used for identifying the parameters is fundamental for creating reasonably accurate prediction models. For this specific approach, training of the predictive models leverages the DoE as discussed in sub-section 8.2.1 by using existing benchmarks such as PARSEC [5], Mediabench-II [32], SPEC 2006 [18], as well as their unique combinations; this training is guided by DoE such that the properties of these experiments are satisfied. These benchmarks and their combinations are used with different parameters, such as levels of parallelization, number of threads, computational load, memory requirements, etc., to excite the platform from different dimensions and systematically collect the response data for training. For instance, PARSEC benchmarks have good Instruction-Level Parallelism (ILP) diversity and are excellent for building predictive models that capture the computing and memory behaviors. However, these benchmark applications are CPU bound and mostly exhibit a constant high load, which may not be ideal for properly evaluating the impact of distinct load contribution patterns. For this reason, a set of synthetic microbenchmarks with attributes that reflect interactive behaviors (I/O dependent applications) and other cross-layer attributes are created and mixed with traditional benchmark suites during the training data collection for the predictive models [38, 46].

The use of specialized microbenchmarks [44, 46] can provide further diversity to accurately capture cross-layer characteristics. For example, in [46] a set of multi-threaded synthetic benchmarks – interactive microbenchmarks (IMB) – enable selective control of the workload, phasic/ bursty behavior, and interactivity (sleep and wait periods). These IMBs can be configured to have throughput (T) and interactivity (I) that control the sleep/wait periods for high(H), medium(M), and low(L) values. Using this approach, a diverse combination of synthetic benchmarks can be generated to stress various dimensions of cross-layer attributes. For instance, the combination "HTHI" represents a high throughput and high interactivity IMB configuration; all other combinations are similarly used in the experiments to capture the cross-layer behavior [38, 46].

8.3.6 Selecting the HMP Configuration

Once the response surface of the system goal is formed using the predictive models, different search heuristics can be used to find the most suitable HMP configuration. As an example, the configuration that performed the best in most cases as the number of threads (or load) increases can be selected by searching the feasible configurations. Other heuristics or optimization criteria can be used to select the configurations from the Pareto front [16, 21].

8.4 Case Study: Experimental Evaluation of Cross-Layer DSE of HMPs

In this section, an example case study of the presented cross-layer approach is illustrated for a contemporary heterogeneous multi-core architecture such as the ARM big.LITTLE [17]), with the goal of quantifying the benefits of different architectural configurations. To emulate different core types, different classes of publicly available Alpha processor models [25, 30] (Table 8.2) are used to construct a realistic HMP model in Gem5 [6] by specifying their multilayer architectural features. Note that the performance of the processors in terms of average IPC, area, and power are normalized with respect to the smallest EV4 (Alpha 21064) core. Also observe that the asymmetric increase of approximately 82× in chip area just to double the performance of an EV8 core with respect to an EV4 core. This asymmetry (or heterogeneity) in scaling is essentially exploited by HMPs to achieve performance, power, and energy efficiency for a given area budget.

To illustrate the CLDSE approach, some experiments can be run by considering chip/die area budget of four Alpha 21264 (EV6) processors as the system-level constraint. Observe that all the possible distinct combinations with three classes of processors (EV4, EV5, and EV6) that meet the area budget are numbered for the 37 possible candidate configurations as shown in Fig. 8.5. To represent a diverse set of workloads, 8 Mediabench-II algorithms [32] and PARSEC benchmarks [5] are selected as representative workloads; their execution time and power consumption are generated using a combination of Gem5 [6] and McPAT[34], respectively, as shown in Fig. 8.6. The performance and power values for each processor core type are generated through full system simulation as shown in Table 8.2. To consider the effect of varying workload and other microarchitectural effects, the number of threads in the benchmark program are varied with different inputs in the cycle accurate full system Gem5 simulation. Here each benchmark is viewed as a single threaded task. The effect of diverse multi-threaded workloads on the

Table 8.2 Alpha processor cores performance, area and power [30]

Alpha core	Issue width	I-cache	D-cache	Branch prediction	# MSHRs	IPC[a]	Area[a]	Peak power(W)	Avg. power(W)	Power[a]
EV4	2	8 KB, DM	8 KB, DM	2 KB, 1-bit	2	1.00	1.00	4.97	3.73	1.00
EV5	4	8 KB, DM	8 KB, DM	2K-, gshare	4	1.30	1.76	9.83	6.88	1.84
EV6	6	64 KB, 2 Way	64 KB, 2 Way	Hybrid, 2 level	8	1.87	8.54	17.8	10.68	2.86
EV8	8	64 KB, 4 Way	64 KB, 4 Way	Hybrid, 2×EV6 size	16	2.14	82.2	92.88	46.44	12.45

[a]Normalized versus EV4; all cores scaled to 0.1 µm, at 2.1 GHz; IPC based on SPEC CPU benchmarks

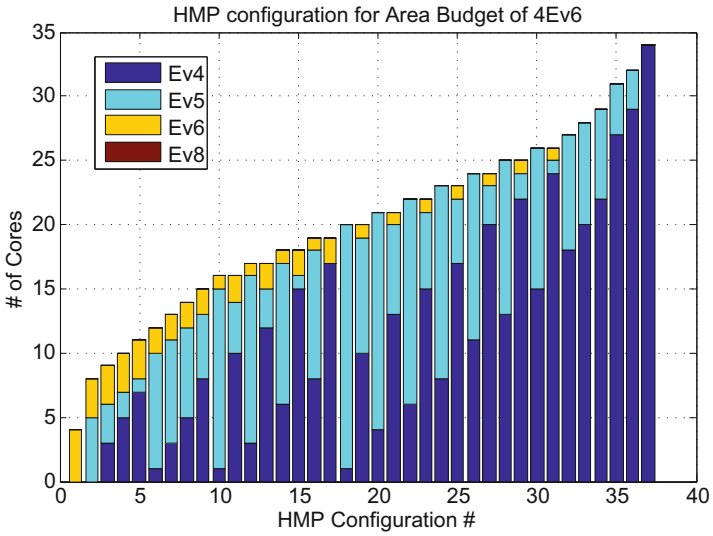


Fig. 8.5 HMP configurations for area budget of 4×EV6. Total of 37 configurations numbered from 1 to 37 from left to right

platform can be simulated by using PARSEC benchmarks or by gradually increasing the number of single threaded tasks and performing the allocation for a given platform. The combination of these benchmarks form new composite tasks (e.g., JPEG compression followed by AES encryption), and the execution time and power consumption of the composite task can be computed as the sum of execution time and power consumption of the individual benchmarks, respectively. This tests architectural configurations with more than 100 cores (e.g., area budget of 4 EV8 results in 46,428 configuration with as many as 330 EV4 cores). With the variability in number of tasks, the objective functions of each architectural combination with system goal min$D$ is shown in Fig. 8.7.

To demonstrate the ability to cover a large design space, an initial set of simulations generated by DoE is used to build the response surface model of the system for the following design objectives: make-span/delay, power, energy, Energy-Delay Product (EDP), and Energy-Delay Square Product (EDSP) as listed in Table 8.1. These initial simulation points are used to construct the predictive models by using linear regression to obtain the coefficients of the expression (8.5) by performing least square fitting of the data. Table 8.3 shows the computational complexity (execution time) and accuracy of the predictors in comparison to a full system simulation (over two orders of magnitude at maximum prediction error of 10%) of the platform for some sample benchmarks. The presented CLDSE shows that an allocation strategy that performs well with one architectural configuration does not perform equally well for another architectural configuration and there is a rich design space to exploit for a specific solution. The predictive models demonstrate

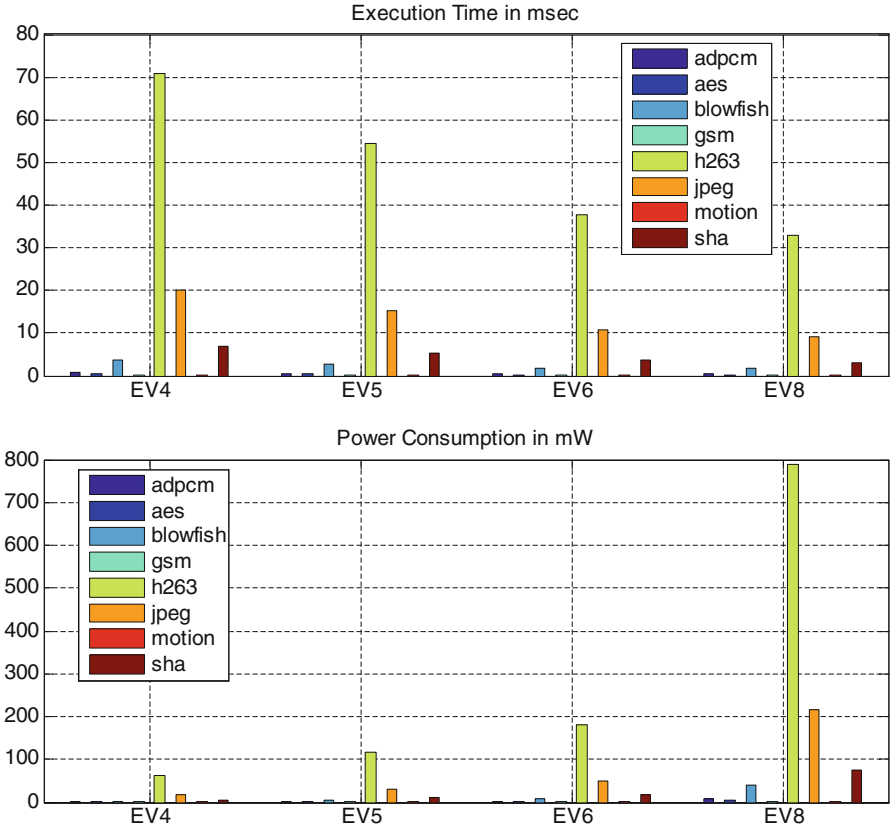


Fig. 8.6 Average power and execution time for eight benchmarks for different Alpha processors

the relative merit for heterogeneous multi-core processor configurations for the same area budget and different allocation strategies. Furthermore, the allocation strategies are compared with a heterogeneity oblivious random allocation with variability in number of task and the execution time as shown in Fig. 8.8. The joint impact of considering the workload variability (with variations in number of tasks and intra-task execution time variations) with allocation strategies shows that almost all the HMP configuration will under-perform by as much as 50 and 70%, respectively, in terms of energy delay product (EDP) and energy delay square product (ED^2) if a heterogeneity agnostic allocation policy (e.g., random policy as in vanilla Linux Kernel) is used. Thus, heterogeneity-aware allocation strategies are crucial for almost any HMP platform configurations, and their impact is significant as the system is loaded with more tasks. This approach can be used to search for the best performing architecture (Table 8.4) as the most preferable architecture (most frequently occurring) for different system goals using a given allocation strategy with the given equal area budget constraints. Observe that for the given area

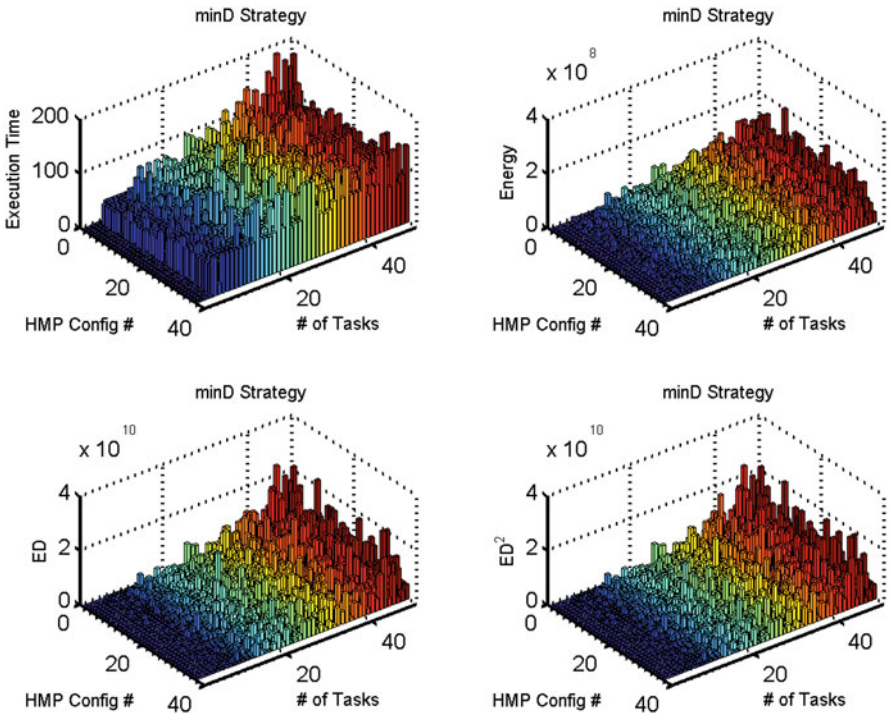


Fig. 8.7 Objectives with variability in number of task for delay only task allocation strategy ($minD$) using the predictive models. Lower is better

Table 8.3 Execution time and prediction model performance on Intel i7 2.4 GHz machines

Benchmarks	HMP config	Gem5 full system simulation time	Prediction model execution time	Prediction error
H.264	#1 (4 cores)	>2 days	< 1 s	<5%
Bodytrack	#2 (8 cores)	>4 days	< 1 s	<5%
Blackscholes	#10 (16 cores)	>7 days	< 1 s	<8%
Fluidanimate	#10 (16 cores)	>7 days	< 1 s	<8%
Mix of above	#36 (32 cores)	>10 days	< 1 s	<10%

budget, the architectural combination #9($8 \times$ EV4, $5 \times$ EV5, $2 \times$ EV6) with 8 EV4 cores, 5 EV5 scores, and 2 EV6 cores has superior performance in terms of EDP and ED^2.

This section has outlined a case study demonstrating the benefits of cross-layer design space exploration of HMPs. These preliminary studies show that much more research is required to exploit this rich and complex space of cross-layer optimizations for emerging heterogeneous architectures.

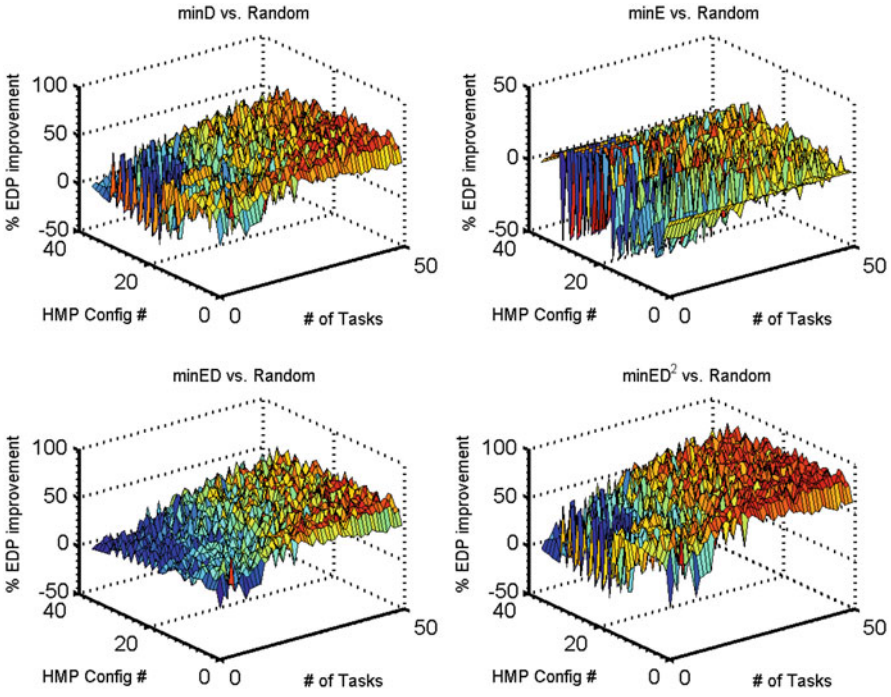


Fig. 8.8 DSE with predictive models: comparison of SA-based static allocation strategies with random allocation strategy (as in vanilla Linux). Higher is better

Table 8.4 Preferred architectural composition with different system goals and allocation strategies

Goals/objective	J_D (min$D$)	J_E (min$E$)	J_{ED}(minED)	J_{ED^2}(minED^2)
$\mathbf{C_{PerfMax}}$	#1 (4 × EV6)	#1(4 × EV6)	#1(4 × EV6)	#1(4 × EV6)
$\mathbf{C_{EnergyMin}}$	#37(34 × EV4)	#9(8 × EV4, 5 × EV5, 2 × EV6)	#37(34 × EV4)	#37(34 × EV4)
$\mathbf{C_{PowerMin}}$	#9(8 × EV4, 5 × EV5, 2 × EV6)	#37(34 × EV4)	#2(5 × EV5, 3 × EV6)	#2(5 × EV5, 3 × EV6)
$\mathbf{C_{EEMax}}$	#9(8 × EV4, 5 × EV5, 2 × EV6)	#37(34 × EV4)	#9(8 × EV4, 5 × EV5, 2 × EV6)	#9(8 × EV4, 5 × EV5, 2 × EV6)

8.5 Conclusions

Cross-layer architectural design space exploration presents significant opportunities for architects to comparatively evaluate design choices early in the design, while accounting for complex interactions between constraints at multiple abstraction

levels. This chapter presented an exemplar case study for the exploration of single-chip, single-ISA heterogeneous multi-core processors.

Recent research has highlighted the potential benefits of single-ISA heterogeneous multi-core processors over cost-equivalent homogeneous ones, and it is likely that future processors will integrate cores that have the same ISA but offer different performance and power characteristics. However, there are few efforts that address the problem of HMP composition, constituting of different core types. This chapter presented a cross-layer (across application, operating system, and hardware architecture layer) approach of single-ISA heterogeneous multi-core processors using predictive models to investigate the interactions and influence of heterogeneity of hardware architectures (configurations, number, and types of cores) and multi-objective allocation strategies along with diverse types of workloads under system-level constraints (such as equal area or power budget). A versatile and realistic approach was outlined along with clear methodology to build cross-layer predictive models of application and system interactions that can be used in the HMP compositions. The presented cross-layer approach quantifies the relative merits of one architectural configuration and allocation strategy over others and helps in selecting most promising heterogeneous architectures. The predictive cross-layer approach enables the chip architect and designer to comparatively evaluate and select the most promising (e.g., energy and performance efficient) HMP configuration in over two order of magnitude less simulation time compared to a full system simulation especially during the early design and verification stages when the design space is at its largest. The approach embodied in this chapter should be applicable for design space exploration of many emerging programmable architectures.

Acknowledgments This work was partially supported by the NSF Variability Expedition award CCF-1029783.

References

1. Anderson MJ, Whitcomb PJ (2000) Design of experiments. Wiley Online Library. doi: 10.1002/0471238961.0405190908010814.a01.pub3. http://onlinelibrary.wiley.com/doi/10.1002/0471238961.0405190908010814.a01.pub3/abstract. Accessed Sep 2010
2. Annamalai A, Rodrigues R, Koren I, Kundu S (2013) An opportunistic prediction-based thread scheduling to maximize throughput/watt in amps. In: 2013 22nd international conference on parallel architectures and compilation techniques (PACT), pp 63–72. doi: 10.1109/PACT.2013.6618804
3. Balakrishnan S et al (2005) The impact of performance asymmetry in emerging multicore architectures. SIGARCH Comput Archit News 33(2):506–517. doi: 10.1145/1080695.1070012
4. Becchi M et al (2006) Dynamic thread assignment on heterogeneous multiprocessor architectures. In: Proceedings of the 3rd conference on computing frontiers, CF '06. ACM, New York, pp 29–40. doi: 10.1145/1128022.1128029
5. Bienia C et al (2008) The parsec benchmark suite: characterization and architectural implications. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques. ACM, pp 72–81

6. Binkert N et al (2011) The gem5 simulator. SIGARCH Comput Archit News 39(2):1–7. doi: 10.1145/2024716.2024718
7. Chen T, Chen Y, Guo Q, Zhou ZH, Li L, Xu Z (2013) Effective and efficient microprocessor design space exploration using unlabeled design configurations. ACM Trans Intell Syst Technol (TIST) 5(1):20
8. Chen J et al (2009) Efficient program scheduling for heterogeneous multi-core processors. In: 46th ACM/IEEE design automation conference, 2009, DAC '09, pp 927–930
9. Chen T, Guo Q, Tang K, Temam O, Xu Z, Zhou ZH, Chen Y (2014) Archranker: a ranking approach to design space exploration. In: 2014 ACM/IEEE 41st international symposium on computer architecture (ISCA). IEEE, pp 85–96
10. Chitlur N, Srinivasa G, Hahn S, Gupta P, Reddy D, Koufaty D, Brett P, Prabhakaran A, Zhao L, Ijih N et al (2012) Quickia: exploring heterogeneous architectures on real prototypes. In: 2012 IEEE 18th international symposium on high performance computer architecture (HPCA). IEEE, pp 1–8
11. Cook H, Skadron K (2008) Predictive design space exploration using genetically programmed response surfaces. In: Proceedings of the 45th annual design automation conference. ACM, pp 960–965
12. Deb K (2001) Multi-objective optimization using evolutionary algorithms. Wiley, Chichester
13. Dubach C, Jones T, O'Boyle M (2007) Microarchitectural design space exploration using an architecture-centric approach. In: Proceedings of the 40th annual IEEE/ACM international symposium on microarchitecture. IEEE Computer Society, pp 262–271
14. Eeckhout L, Vandierendonck H, Bosschere K (2002) Workload design: selecting representative program-input pairs. In: Proceedings of the 2002 international conference on parallel architectures and compilation techniques. IEEE, pp 83–94
15. Ge R et al (2010) Powerpack: Energy Profiling and analysis of high-performance systems and applications. IEEE Trans Parallel Distrib Syst 21(5):658–671. doi: 10.1109/TPDS.2009.76
16. Givargis T, Vahid F, Henkel J (2002) System-level exploration for pareto-optimal configurations in parameterized system-on-a-chip. IEEE Trans Very Large Scale Integr Syst 10(4):416–422
17. Greenhalgh P (2011) Big.little processing with arm cortex-a15 & cortex-a7: improving energy efficiency in high-performance mobile platforms. http://www.arm.com/files/downloads/big.LITTLE_Final.pdf
18. Henning JL (2006) Spec cpu2006 benchmark descriptions. ACM SIGARCH Comput Archit News 34(4):1–17
19. Hill M et al (2008) Amdahl's law in the multicore era. Computer 41(7):33–38. doi: 10.1109/MC.2008.209
20. İpek E, McKee SA, Caruana R, de Supinski BR, Schulz M (2006) Efficiently exploring architectural design spaces via predictive modeling. SIGPLAN Not 41(11):195–206. doi: 10.1145/1168918.1168882
21. Ipek E, McKee SA, Singh K, Caruana R, Supinski BRd, Schulz M (2008) Efficient architectural design space exploration via predictive modeling. ACM Trans Archit Code Optim (TACO) 4(4):1
22. Jin Z, Cheng AC (2008) Improve simulation efficiency using statistical benchmark subsetting: an implantbench case study. In: Proceedings of the 45th annual design automation conference. ACM, pp 970–973
23. Joseph P, Vaswani K, Thazhuthaveetil MJ (2006) Construction and use of linear regression models for processor performance analysis. In: The twelfth international symposium on high-performance computer architecture. IEEE, pp 99–108
24. Kenzo VC et al (2012) Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In: International symposium on computer architecture, ISCA'12
25. Kessler R (1999) The alpha 21264 microprocessor. IEEE Micro 19(2):24–36. doi: 10.1109/40.755465
26. Keutzer K et al (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543. doi: 10.1109/43.898830

27. Khan S, Xekalakis P, Cavazos J, Cintra M (2007) Using predictivemodeling for cross-program design space exploration in multicore systems. In: Proceedings of the 16th international conference on parallel architecture and compilation techniques. IEEE Computer Society, pp. 327–338
28. Kleijnen JP (2008) Design and analysis of simulation experiments, vol 20. Springer, New York/London
29. Kumar R et al (2004) Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: Proceedings of the 31st annual international symposium on computer architecture, pp 64–75. doi: 10.1109/ISCA.2004.1310764
30. Kumar R et al (2005) Heterogeneous chip multiprocessors. Computer 38(11):32–38. doi: 10.1109/MC.2005.379
31. Lee BC, Brooks DM (2006) Accurate and efficient regression modeling for microarchitectural performance and power prediction. In: ACM SIGPLAN notices, vol 41. ACM, pp 185–194
32. Lee C, Potkonjak M, Mangione-Smith WH (1997) Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In: Proceedings of the 30th annual ACM/IEEE international symposium on microarchitecture. IEEE Computer Society, pp 330–335
33. Lee BC, Collins J, Wang H, Brooks D (2008) Cpr: composable performance regression for scalable multiprocessor models. In: 2008 41st IEEE/ACM international symposium on microarchitecture, 2008, MICRO-41. IEEE, pp 270–281
34. Li S et al (2009) Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: 42nd annual IEEE/ACM international symposium on microarchitecture, 2009, MICRO-42, pp 469–480
35. Liu HY, Carloni LP (2013) On learning-based methods for design-space exploration with high-level synthesis. In: Proceedings of the 50th annual design automation conference. ACM, p 50
36. Liu G et al (2013) Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In: 2013 IEEE 31st international conference on computer design (ICCD), pp 54–61. doi: 10.1109/ICCD.2013.6657025
37. Montgomery DC: Design and analysis of experiments. Wiley, Hoboken (2008)
38. Mück T, Sarma S, Dutt N (2015) Run-DMC: runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency. In: Proceedings of the 10th international conference on hardware/software codesign and system synthesis. IEEE, pp 173–182
39. NVidia (2011) Variable SMP – a multi-core CPU architecture for low power and high performance. http://www.nvidia.cn/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-\Architecture-for-Low-Power-and-High-Performance-v1.1.pdf
40. Palermo G, Silvano C, Zaccaria V (2009) Respir: a response surface-based pareto iterative refinement for application-specific design space exploration. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(12):1816–1829. doi: 10.1109/TCAD.2009.2028681
41. Phansalkar A, Joshi A, John LK (2007) Subsetting the spec CPU2006 benchmark suite. ACM SIGARCH Comput Archit News 35(1):69–76
42. Pimentel A et al (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Computers 55(2):99 – 112. doi: 10.1109/TC.2006.16
43. Santner TJ, Notz W, Williams B (2003) The design and analysis of computer experiments. Springer, New York
44. Sarma S (2016) Cyber-physical-system-on-chip (CPSoC): an exemplar self-aware SoC and smart computing platform
45. Sarma S, Dutt N (2015) Cross-layer exploration of heterogeneous multicore processor configurations. In: 2015 28th international conference on VLSI design (VLSID), pp 147–152. doi: 10.1109/VLSID.2015.30
46. Sarma S, Muck T, Bathen LAD, Dutt N, Nicolau A (2015) Smartbalance: a sensing-driven linux load balancer for energy efficiency of heterogeneous mpsocs. In: Proceedings of the 52nd annual design automation conference, DAC '15. ACM, New York, pp 109:1–109:6. doi: 10.1145/2744769.2744911
47. Shelepov D et al (2009) Hass: a scheduler for heterogeneous multicore systems. SIGOPS Oper Syst Rev 43(2):66–75. doi: 10.1145/1531793.1531804

48. Teodorescu R, Torrellas J (2008) Variation-aware application scheduling and power management for chip multiprocessors. SIGARCH Comput Archit News 36(3):363–374. doi: 10.1145/1394608.1382152
49. Vidyarthi DP et al (2009) Scheduling in distributed computing systems: analysis, design & models, a research monogram. Springer
50. Wu W, Lee BC (2012) Inferred models for dynamic and sparse hardware-software spaces. In: Proceedings of the 2012 45th annual IEEE/ACM international symposium on microarchitecture. IEEE Computer Society, pp 413–424
51. Yi JJ, Lilja DJ, Hawkins DM (2003) A statistically rigorous approach for improving simulation methodology. In: Proceedings of the ninth international symposium on high-performance computer architecture, 2003, HPCA-9 2003. IEEE, pp 281–291

Scenario-Based Design Space Exploration

9

Andy Pimentel and Peter van Stralen

Abstract

Modern embedded systems are becoming increasingly multifunctional, and, as a consequence, they more and more have to deal with dynamic application workloads. This dynamism manifests itself in the presence of multiple applications that can simultaneously execute and contend for resources in a single embedded system as well as the dynamic behavior within applications themselves. Such dynamic behavior in application workloads must be taken into account during the early system-level Design Space Exploration (DSE) of Multiprocessor System-on-Chip (MPSoC)-based embedded systems. *Scenario-based DSE* utilizes the concept of application scenarios to search for optimal mappings of a multi-application workload onto an MPSoC. To this end, scenario-based DSE uses a multi-objective genetic algorithm (GA) to identify the mapping with the best average quality for all the application scenarios in the workload. In order to keep the exploration of the scenario-based DSE efficient, fitness prediction is used to obtain the quality of a mapping. This fitness prediction implies that instead of using the entire set of all possible application scenarios, a small but representative subset of application scenarios is used to determine the fitness of mapping solutions. Since the representativeness of such a subset is dependent on the application mappings being explored, these representative subsets of application scenarios are dynamically obtained by means of coexploration of the scenario subset space. In this chapter, we provide an overview of scenario-based DSE and, in particular, present multiple techniques for fitness prediction using representative subsets of application scenarios: a stochastic, deterministic, and hybrid combination.

A. Pimentel (✉)
University of Amsterdam, Amsterdam, The Netherlands
e-mail: a.d.pimentel@uva.nl

P. van Stralen
Philips Healthcare, Best, The Netherlands
e-mail: peter.van.stralen@philips.com

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_10

Acronyms

ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
DSE	Design Space Exploration
ESL	Electronic System Level
FIFO	First-In First-Out
FS	Feature Selection
GA	Genetic Algorithm
JPEG	Joint Photographic Experts Group
KPN	Kahn Process Network
MJPEG	Motion JPEG
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
SBS	Sequential Backward Selection
SFS	Sequential Forward Selection

Contents

9.1 Introduction

To cope with the design complexities of *Multi-Processor System-on-Chip* (MPSoC)-based embedded systems [15], Electronic System Level (ESL) design [6, 12] has become a promising approach for raising the abstraction level of design and thereby increasing the design productivity. Early design space exploration (DSE) is an important ingredient of such ESL design, which has received significant research attention in recent years [8, 10, 19]. The majority of all these DSE efforts still evaluates and explores MPSoC architectures under single-application workloads. This is, however, increasingly unrealistic since modern embedded devices, especially in the consumer electronics domain, are nowadays highly multifunctional and feature dynamic application workloads. For example, a mobile

phone has become a multimedia device that is not only used for calling, but it is also connected to the Internet and has become a decent camera. With the increased capabilities of current smartphones, they have almost the same possibilities as desktop computers. Another trend is to make consumer devices "smart." Smart digital cameras are able to directly share photos on the Internet. Photos can be edited on the camera and can also be tagged with a GPS location. Similarly, smart televisions also enhance the functionality of the television as they not only show an incoming video stream from a decoder but can, e.g., also show photos from a memory card or install additional applications. This trend of smartness does not only increase the number of applications but also the dynamism of the application workloads on these embedded systems. For old analogue televisions with CRT, the characteristics of the incoming video stream were exactly known: the size of the frames, the frame rate, etc. Currently, however, these streams become more dynamic: High-definition (HD) television may, for example, have varying frame rates and frame sizes. Additionally, 3D television may double the number of frames that need to be decoded.

This chapter will therefore use the concept of application scenarios [7, 17] to introduce *scenario-based DSE* [28, 30]. Application scenarios are able to describe the dynamism of embedded applications and the interaction between the different applications on the embedded system. The concept of application scenarios is illustrated in Fig. 9.1. An application scenario consists of two parts: an inter- and an intra-application scenario. An *inter-application scenario* describes the interaction between multiple applications, i.e., which applications are concurrently executing at a certain moment in time. Inter-application scenarios can be used to prevent the overdesign of a system. If some of the applications cannot run concurrently, then there is no need of reserving resources for the situation where these applications are running together. *Intra-application scenarios*, on the other hand, describe the different execution modes (or operation modes) for each individual application. In the example application scenario in Fig. 9.1, the left-hand side shows the selected

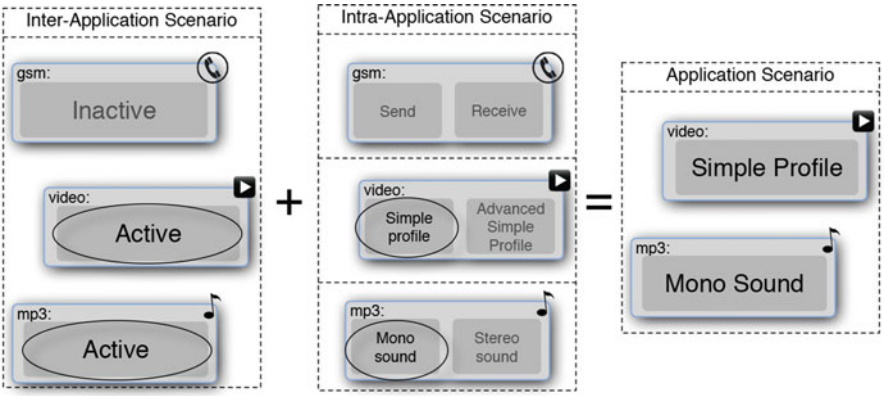


Fig. 9.1 An illustration of application scenarios

inter-application scenario. In this case, the video and the MP3 applications are active, while the GSM application is inactive. In the middle, the intra-application scenarios are shown. The video application can, for example, decode video using a simple profile and an advanced simple profile. For the intra-application scenario, it is decided to decode video using a simple profile and to play mono music with the MP3 application. As the GSM is inactive, no operation mode needs to be selected for the GSM application. Hence, the application scenario is the sum of the inter- and intra-application scenarios: the video application is decoding using a simple profile, and the MP3 application is playing music in a mono sound.

9.2 Application Dynamism

To illustrate the consequences of dynamic application behavior in terms of extra-functional aspects (like system performance and power consumption), this section presents a small, motivational case study in which a Motion JPEG (MJPEG) decoder application – with different intra-application scenarios – is mapped onto a heterogeneous bus-based MPSoC architecture with four processors and a single shared memory. First, we have randomly picked three mappings of the MJPEG application on our bus-based architecture. For each of these mappings, we used the Sesame system-level MPSoC simulation framework [3, 18] to determine the fitness values (in this case, execution time and power consumption) for each *individual intra-application scenario*. The resulting fitness values of these mappings are shown in Fig. 9.2, where the horizontal and vertical axes refer to execution time and

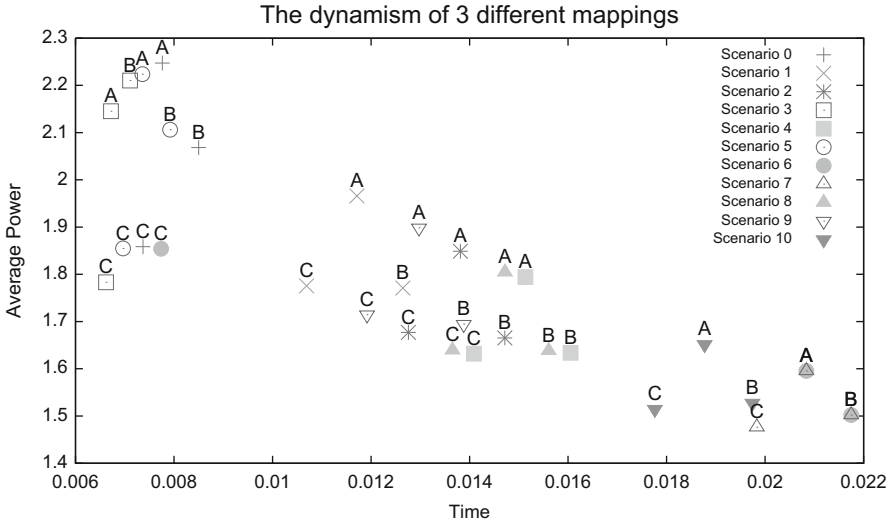


Fig. 9.2 An illustration of the dynamism of different scenarios. For three different mappings, the fitness of each individual scenario is shown

power consumption, respectively. These values are only used to compare different mappings and intra-application scenarios. Therefore, they do not have a unit. In this graph, the letters A–C are the different mappings, whereas 0–10 are the different intra-application scenarios of the MJPEG application.

Irrespective of the mapping, scenario 7 is the intra-application scenario that consumes the least amount of power. The scenario with the highest power consumption, on the other hand, depends on the mapping. For mappings A and C, scenario 0 has the highest power consumption. In case of mapping B, however, scenario 3 has the highest power consumption. To explain this behavior, both the scenarios and the mappings must be analyzed. Scenario 3 involves the decoding of a frame that has a much higher compression ratio than scenario 0. As a consequence, scenario 3 requires less communication than scenario 0. Moreover, for mapping B, the shared bus is fully utilized, while in mappings A and C, there is still some capacity left on the shared bus. As a result, the reduction in communication between scenarios 0 and 3 has more effect on the execution time of mapping B than it has on mappings A and C. Although the consumed energy for scenario 3 is lower than the consumed energy for scenario 0 for all of the three mappings, the larger difference in execution time results in higher power consumption for mapping B.

Figure 9.2 also shows other interesting behavior with respect to scenario 6. For example, scenario 6 with mappings A and B has the same fitness as scenario 7. For mapping C, however, the execution time of scenario 6 is much lower than for scenario 7. Without going into details, scenario 6 would lead to the conclusion that mapping C consumes more power than mapping A. This conclusion contrasts with the conclusion that can be drawn from comparing the power consumption using the other scenarios (i.e., the power consumption of mapping C is lower than mapping A). In a sense, scenario 6 gives a deceiving view of the quality-ordering relation between the different mappings.

To provide more insight into this problem, the potential Pareto dominance relations between the mappings of the experiment illustrated in Fig. 9.2 are listed in Table 9.1. See ▸ Chaps. 6, "Optimization Strategies in Design Space Exploration"

Table 9.1 The Pareto dominance relations comparing the mappings from Fig. 9.2 for each individual scenario. The symbol || stands for incomparable fitness values

Scenario	$A \ldots B$	$A \ldots C$	$B \ldots C$	Front
0	||	$>$	$>$	C
1	||	$>$	||	B, C
2	||	$>$	||	B, C
3	$\leq$	$>$	$>$	C
4	||	$>$	$>$	C
5	||	$>$	$>$	C
6	||	||	||	A, B, C
7	||	$>$	$>$	C
8	||	$>$	||	B, C
9	||	$>$	||	B, C
10	||	$>$	$>$	C

and ▸ 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration" for a discussion on Pareto dominance. In the three columns in the middle of the list in Table 9.1, the unique mapping comparisons are shown: mapping A versus B, mapping A versus C, and mapping B versus C. Next, for each individual intra-application scenario, the fitness values for the different mappings are compared. In this way, three different types of relations are obtained: (1) a mapping is equal to or fully dominates the other mapping ($\leq$), (2) a mapping is dominated by another mapping ($>$), and (3) the mappings are not comparable using the Pareto dominance relation ($||$). Finally, the last column shows the Pareto front based on the fitness values of the specific intra-application scenario.

As a first observation, one can see that for none of the relations, it is the case that all scenarios fully agree on the type of the relation. For the first two relations (where mapping A is compared with mappings B and C), only one scenario differs with respect to the relation type. In case of the comparison between mapping A and mapping B, the fitness values for most of the scenarios determine that mapping A is incomparable with mapping B. Only the fitness values of scenario 3 lead to a different conclusion: mapping A dominates mapping B. Similarly, mapping C dominates mapping A for most of the intra-application scenarios. Only the fitness values of scenario 6 are incomparable for mappings A and C.

The problem arises with the relation between mapping B and mapping C. Judging on 6 out of the 11 scenarios, mapping B is better than mapping C. Based on the other five scenarios, however, one comes to the conclusion that mapping B is incomparable with mapping C. These kinds of uncertainties complicate the scenario-based DSE. The DSE ends up with a Pareto front, but not all the intra-application scenarios agree on what the Pareto front should be. In the example of Table 9.1, three different Pareto fronts are observed, from which the front with only mapping C is the most common.

Based on the most common Pareto front with only mapping C, one could conclude that the set $\{0, 3, 4, 5, 7, 10\}$ of intra-application scenarios is *representative* for the MJPEG application. This representativeness, however, is completely dependent on which mappings are evaluated. In case only mappings A and B would have been taken into account, intra-application scenario 3 would have been interpreted as an unrepresentative scenario. However, if this scenario 3 is excluded for the comparison between mapping B and mapping C, there is no majority anymore for one of the Pareto dominance relation types. In the next section, which introduces our scenario-based DSE framework, we will explain how we deal with the above problem.

9.3 Scenario-Based DSE Framework

Conceptually, scenario-based DSE [28, 30] is an exploration technique for embedded systems with a dynamic multi-application workload. In this chapter, an exploration framework for scenario-based DSE is presented that aims to provide a *static mapping* of a multi-application workload onto an MPSoC. The mapping is to be used during the entire lifetime of an embedded system. Consequently, the

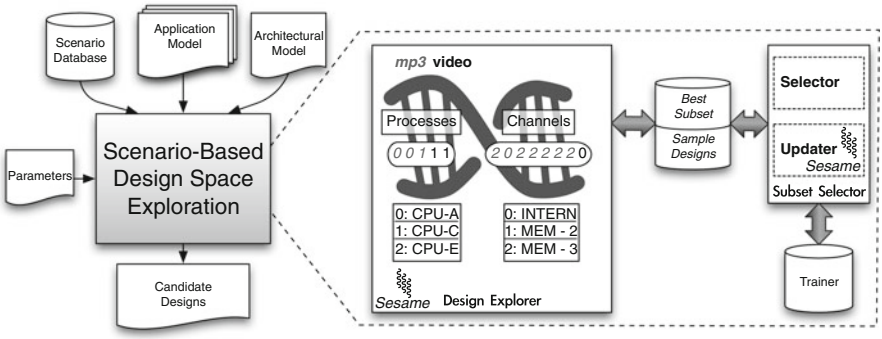


Fig. 9.3 The exploration framework for scenario-based DSE

average behavior of the designed MPSoC must be as good as possible for all the different application scenarios. Currently, we assume an equal likelihood for each application scenario. The approach, however, can easily be generalized to different probability distributions.

In this work, we assume that a multi-application workload mapping implicitly consists of two aspects: (1) *allocation* and (2) *binding*. The allocation selects the architectural components for the MPSoC platform architecture. Only the selected components will be deployed on the MPSoC platform. These components include processors, memories, and supporting interconnects like communication buses, crossbar switches, and so on. Subsequently, the binding specifies which application task or application communication is performed by which (allocated) MPSoC component.

Figure 9.3 shows the exploration framework. The left part of the picture provides a general flow, whereas the right part illustrates the scenario-based DSE in more detail. As an input, the scenario-based DSE requires a scenario database, application models, and an MPSoC platform architecture model. Binding is performed for a multi-application workload, and the description of this workload is split into two parts: (1) the structure and (2) the behavior. The structure of applications is described using application models. For these models, the *Kahn Process Network* (KPN) model of computation is used [11], which models applications as a network of concurrent processes communicating via FIFO channels (see also refer to the section "Overview of Basic Data Flow Models" in the ▸ Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign"). Next to the KPN application models, a scenario database [29] explicitly stores all the possible multi-application workload behaviors in terms of application scenarios (i.e., intra- and inter-application scenarios).

An important problem that needs to be solved by scenario-based DSE is the fact that the number of possible application scenarios is too large for an exhaustive evaluation of *all* – or even a restricted set of – the design points with *all* the scenarios during the MPSoC DSE. Therefore, a small but *representative subset of scenarios* must be selected for the evaluation of MPSoC design points. This representative

subset must compare mappings and should lead to the same performance ordering as would have been produced when the complete set of the application scenarios would have been used. However, the selection of such a representative subset is not trivial, as was already explained in Sect. 9.2 and studied in more detail in [27]. This is because the representative subset is dependent on the current set of mappings that are explored. Depending on the set of mappings, a different subset of application scenarios may reflect the relative mapping qualities of the majority of the application scenarios.

As a result, the representative subset cannot statically be selected. For a static selection, one would need to have a large fraction of the mappings that are going to be explored during the MPSoC DSE. However, since these mappings are only available during DSE, a dynamic selection method must be used. Thus, both the set of optimal mappings and the representative subset of scenarios need to be *coexplored simultaneously* such that the representative subset is able to adapt to the set of mappings that are currently being explored.

In the scenario-based exploration framework (see Fig. 9.3), two separate components are shown that simultaneously perform the coexploration tasks: the design explorer searches for the set of optimal mappings, while the subset selector tries to select a representative subset of scenarios. As these components are running asynchronously, a shared-memory interface is present to exchange data. For the design explorer, a sample of the current mapping population is stored in the shared memory, whereas the subset selector makes the most representative subset available for the fitness prediction in the design explorer. One of the main advantages of the strict separation of the execution of the design explorer and the subset selector is that the running time of the design explorer becomes more transparent. From a user perspective, this is the most important component, as it will identify the set of optimal mappings.

9.4 Design Explorer

In this section, the design explorer, the component that is responsible for identifying promising mappings, is described. First, our system model is described. This system model formally describes both the applications and the architecture. Next, the system model is used to describe the complete mapping procedure that will be applied during the search for good mappings, which has been implemented using a Genetic Algorithm (GA).

9.4.1 System Model

Our system model is based on the popular Y-chart design approach [13] which implies that we separate *application models* and *architecture (performance) models* while also recognizing an explicit *mapping step* (or layer) to map application tasks onto architecture resources [24]. The system model has been implemented in the Sesame system-level MPSoC simulation framework [3,18], as illustrated in Fig. 9.4.

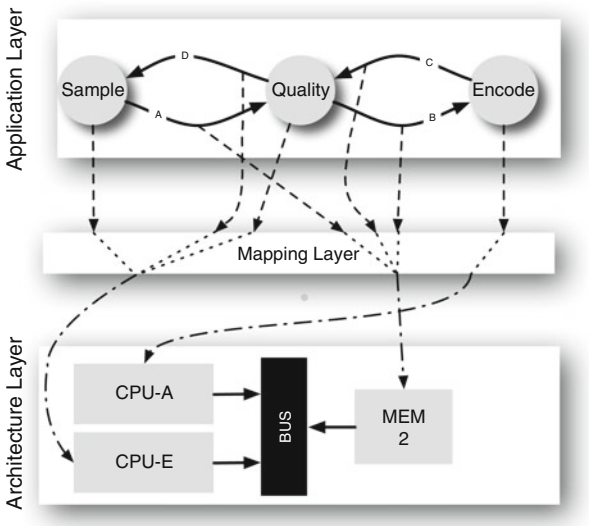


Fig. 9.4 The different layers in the Sesame model and their connections

For a more detailed discussion of the Sesame framework, we also refer the interested reader to chapter ▸ Chap. 30, "DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips". The system model formalizes each of the application, mapping, and architecture layers:

Application (model) layer The application model describes each individual application as a Kahn Process Network (KPN) [11]. A KPN is formally defined as a directed graph $G_K(V, E_K)$. The vertexes V are the process nodes. In Fig. 9.4, an example application model is depicted in which the set V is equal to {SAMPLE, ENCODE, QUALITY}. The communication channels of the application are represented by directed edges $E_K = V \times V$. If, for example, (SAMPLE, QUALITY) is defined in E_K, it means that there is a communication channel from SAMPLE to QUALITY.

Architecture (model) layer The architecture is represented by a directed graph $G_R(R, E_R)$. In this case, the set R contains the architectural components like processors, communication buses, crossbar switches, FIFO buffers, and memories. Edges $E_R = R \times R$, on the other hand, describe the communication links in the architecture.

There are three types of architectural elements: (1) processors, (2) buffers, and (3) interconnects. The processors $R_P \subset R$ are architectural elements that are capable of executing processes. Buffers $R_B \subset R$ are the FIFO buffers/memories used for the communication between the different processors. If two communicating processes are mapped onto the same processor, the communication may also be done internally. This is only possible when a processor supports internal communication. If a processor $p \in R_P$ supports internal communication, a buffer $b \in R_B$ is added to the architecture. Additionally, the buffer is connected to enable

reading and writing of data: $(p, b) \in E_R \wedge (b, p) \in E_R$. Finally, the set of interconnects $R_I \subset R$ is purely meant to connect the various system components.
In the example of Fig. 9.4, the architectural processors R_P consist of CPU-A and CPU-E. Next, the FIFO buffers in the architecture are MEM-2 and CPU-E. This means that CPU-E supports internal communication. Finally, the BUS belongs to the set of interconnects. There are eight edges in the architecture. Six of these links are connected to the bus (for reading and writing). Additionally, the internal communication buffer of CPU-E is connected to the processor for reading and writing. In this example, all communication links are bidirectional.

Mapping Layer The mapping layer connects the application layer to the architecture layer. Hence, it contains only edges: computation edges and communication edges. Computation mapping edges E_X assign KPN processes to the architectural resources. To be precise, the edge $(v, p) \in E_X$ assigns KPN process $v \in V$ to processor $p \in R_P$. A KPN process can only be mapped on processing elements that are feasible of running that task:

$$(v, p) \in E_X \iff \text{Feasible}(p, v) \tag{9.1}$$

This allows for modeling processors that range from general-purpose processors (i.e., those processors $p \in R_P$ for which holds that $\forall v \in V$: Feasible(p, v)) to processors that are able to perform only a limited set of tasks like, e.g., ASICs. Here, we would like to note that it is also possible to map multiple KPN processes onto a single processor if the (modeled) processor type allows this (e.g., in the case of a general-purpose processor, ASIP, etc.). The communication is mapped using communication edges E_C. A communication edge (c, b) maps FIFO channel $c \in E_K$ to FIFO buffer $b \in R_B$.

9.4.2 Mapping Procedure

While the application and the architecture layers are predefined before a DSE is started, the mapping layer is the part of the MPSoC design that needs to be optimized. As discussed before, the mapping consists of two steps: allocation and binding. Allocation can reduce the resource usage of the MPSoC design, whereas the binding maps all processes and channels on the allocated resources. The procedure is as follows:

Allocation First, the architecture resources are selected to use in the allocation α. All types of architecture resources are selected at once: $\alpha_P = \alpha \cap R_P$, $\alpha_B = \alpha \cap R_B$, and $\alpha_I = \alpha \cap R_I$. More precisely, the allocation α contains a subset of resources such that $\alpha \subseteq R$:

$$\left(\sum_{r \in \alpha} \text{area}(r) \right) \leq \text{MAX\_AREA} \tag{9.2}$$

Hence, Eq. 9.2 implies that the total area of the allocated resources may not be larger than the maximal area of the chip. Part of the system model is also the feasibility of the mapping: for each of the processes, there must be at least one processor that is capable of executing the specific process. This is defined as follows:

$$\forall v \in V : |p \in \alpha_P : \text{Feasible}(p, v)| \geq 1 \tag{9.3}$$

Once the allocation α is known, a set of potential communication paths $\psi = (\alpha_P \times \alpha_B \times \alpha_P)$ can be defined:

$$\psi = \{(p_1, b, p_2) : \text{PATH}_\alpha(p_1, b) \wedge \text{PATH}_\alpha(b, p_2)\} \tag{9.4}$$

The set of communication paths ψ is the set of paths such that (:) there is a path from processor p_1 to buffer b and a path from buffer b to processor p_2 (Eq. 9.4). This path may span multiple resources as long as they are allocated:

$$\text{PATH}_\alpha(r_1, r_2) := (r_1, r_2) \in E_R \vee \tag{9.5a}$$

$$\exists r_i \in \alpha_I : (r_1, r_i) \in E_R \wedge \text{PATH}_\alpha(r_i, r_2) \tag{9.5b}$$

The PATH function is recursively defined. There is a path between resources $r_1 \, and \, r_2$ if there is a direct connection between them (Eq. 9.5a). An interconnect r_i can also be used as part of the path. In this case, there must be a direct connection between resource r_1 and interconnect r_i and a path between interconnect r_i and resource r_2 (Eq. 9.5b). An allocation is only valid if there is at least one communication path between each set of processors:

$$\forall p_1, p_2 \in \alpha_P : \exists (p_1, b, p_2) \in \psi \tag{9.6}$$

By enforcing at least a single communication path between each set of processors, the automatic exploration of mappings is guaranteed to find at least one valid mapping. As will be explained later, the procedure randomly picks the processors after which one of the communication paths is selected.

Binding Binding maps all the KPN process nodes onto the allocated resources. There are two steps: (1) computational binding and (2) communication binding. Computational binding β_X maps the processes onto the processors such that $\beta_X \in E_X$:

$$\forall v \in V : |\{p : (v, p) \in \beta_X \wedge p \in \alpha_P\}| = 1 \tag{9.7}$$

Equation 9.7 enforces that each process v is mapped on exactly one allocated processor p. After the computational binding, the communication binding can be done. Recall from Eq. 9.6 that we have enforced that between each set of processors, at least one communication path is present in ψ. Therefore, for each

communication channel in the application, a communication path in the allocated architecture can be selected. More strictly, for each communication channel in the application, an architectural buffer is selected such that $\beta_C \in E_C$:

$$\forall (v_1, v_2), b \in \beta_C : (v_1, p_1) \in \beta_X \wedge \tag{9.8a}$$

$$(v_2, p_2) \in \beta_X \wedge \tag{9.8b}$$

$$(p_1, b, p_2) \in \psi \tag{9.8c}$$

$$\forall c \in E_K : |\{b : (c, b) \in \beta_C\}| = 1 \tag{9.9}$$

Multiple conditions must be enforced. First, the architectural buffer b on which the communication channel (v_1, v_2) of an application is mapped must be a valid communication path. This means that processes v_1 and v_2 must be mapped on processors p_1 and p_2 (Eq. 9.8a and 9.8b) and that (p_1, b, p_2) is within the set of communication paths ψ (Eq. 9.8c). Processor p_1 and processor p_2 do not necessarily need to be different as both of the processes in the communication link may be mapped onto the same processor. Next, all communication channels c (which is a tuple of the two communication processes) must be mapped on exactly one buffer (Eq. 9.9).

A mapping m is the combination of an allocation α and the bindings β_X and β_C. It is only valid if all the preceding constraints (Eqs. 9.2, 9.3, 9.6, 9.7, and 9.9) are fulfilled.

9.4.3 Exploring Mappings Using a Genetic Algorithm

Our aim is to optimize the mapping of an embedded system. Hence, the space of possible mappings must be explored as efficiently as possible. For this purpose, an NSGA-II-based multi-objective GA [4] is used (see also ▸ Chap. 6, "Optimization Strategies in Design Space Exploration"). Figure 9.5 shows the chromosome design for exploring the mapping space. The mapping chromosome consists of two parts: (1) a KPN process part and (2) a KPN communication channel part. Within these parts, all of the applications are encoded consecutively. The gene values encode the architecture components on which the elements of the applications are mapped: the KPN processes are mapped onto processors, and the KPN channels are mapped onto memories. A special memory is the internal memory, as was previously explained.

The example chromosome in Fig. 9.5 has 11 genes. Five genes are dedicated to the processes, and six genes are dedicated to the communicational channels. As there are three potential processors, the gene value for the KPN process part is between 0 and 2. For the memories, there are three possibilities: two memories and a reserved entry for the internal memory. In this way, the binding to architectural components is encoded for all of the processes and channels. The first process gene

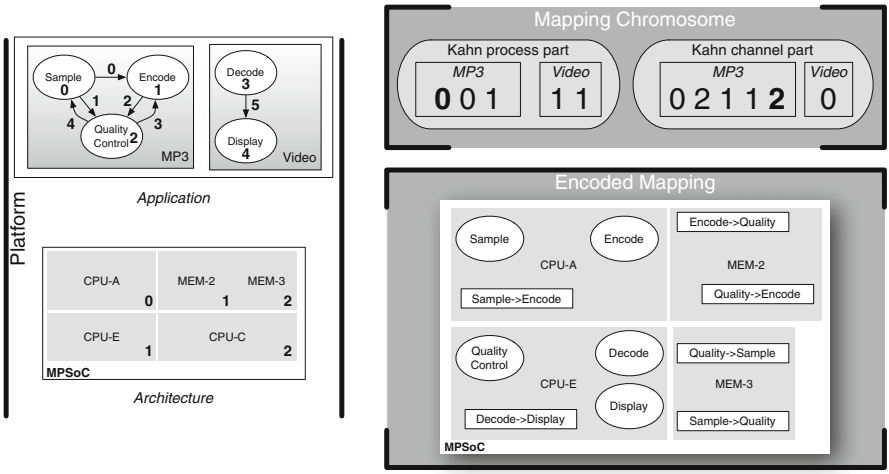


Fig. 9.5 Chromosome representation of a mapping. Both the gene sequence is shown and the mapping that is encoded by the gene sequence

of the MP3 application, for example, has gene value 0. Looking at the platform, gene 0 is the SAMPLE process. This process is mapped on the first processor: CPU-A. Similarly, the channel 4 (QUALITY → SAMPLE) is mapped on MEM-3. The complete encoded mapping is illustrated in Fig. 9.5.

The NSGA-II GA is an elitist selection algorithm that applies non-dominated sorting to select the offspring individuals. Non-dominated sorting ranks all the design points based on their dominance depth [2]. Conceptually, the dominance depth is obtained by iteratively removing the Pareto front from a set of individuals. After each iteration, the rank is incremented. An example is shown in Fig. 9.8c. The main reason for choosing an NSGA-II-based GA is because of its use of the dominance depth for optimization. As will be discussed in Sect. 9.5.2, the dominance depth can easily be used for rating the quality of the representative subset of scenarios.

The dominance of the individuals is based on their fitness. As discussed before, the predicted fitness is used instead of the real fitness. Let S be the total set of scenarios and $\tilde{S}_j$ the representative subset of scenarios at time step j. The fitness objectives of a mapping m are as follows:

$$F(m) = \frac{1}{|S|} \sum_{s \in S} (\text{time}(m,s), \text{energy}(m,s), \text{cost}(m)) \tag{9.10}$$

$$\tilde{F}_{\tilde{S}_j}(m) = \frac{1}{|\tilde{S}_j|} \sum_{s \in \tilde{S}_j} (\text{time}(m,s), \text{energy}(m,s), \text{cost}(m)) \tag{9.11}$$

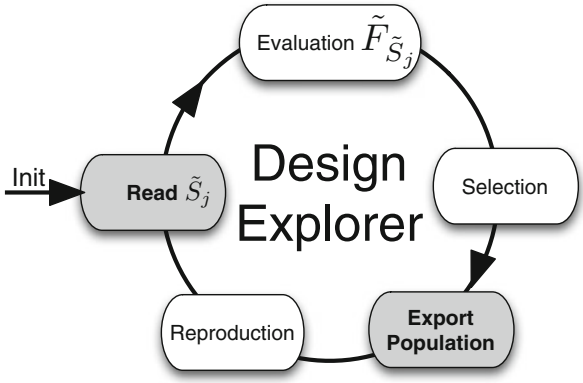


Fig. 9.6 The genetic algorithm for the design explorer extended with the required steps to communicate with the subset selector. The steps that are emphasized involve communication

Given the mapping m and the application scenario s, the functions time(m, s) for execution time and energy(m, s) for energy consumption are evaluated using the Sesame system-level MPSoC simulation framework. The cost of a mapping is independent of the scenario and can be determined statically by adding up the costs of the individual elements within the allocation α. There is an important difference between the *real* fitness F and the *estimated* fitness $\tilde{F}_{\tilde{S}_j}$. The real fitness uses all possible application scenarios to determine the fitness, whereas the estimated fitness only uses a (representative) subset of the scenarios ($\tilde{S}_j \subseteq S$). As a result, the real fitness is independent of the current generation. The predicted fitness, on the other hand, may vary over the different generations. The fitness $\tilde{F}_{\tilde{S}_j}$ is only valid for generation j as the representative subset $\tilde{S}_{j+1}$ may change over time.

In order to update the representative subset of scenarios between the generations, the GA of the design explorer must be extended to support the communication between the design explorer and the subset selector. This extension is shown in Fig. 9.6. Before any individual (i.e., mapping) can be evaluated, the currently most representative subset of scenarios $\tilde{S}_j$ must be acquired. Using the representative subset of scenarios, the design explorer can quickly predict the fitness of all the individuals in the population. This means that depending on the number of changed scenarios in the representative subset of scenarios since the previous generation, the parent population also must be partially reevaluated. This predicted fitness is used to select the individuals for the next generation. In case the scenario subset is representative, the decisions made by the NSGA-II selector are similar to those where the real fitness would have been used. If this is not the case, the scenario subset should be improved. For this purpose, the selected population is exported to the subset selector. Finally, reproduction is performed with the selected individuals. During reproduction, a new population of individuals is generated for usage in the next generation.

9.5 Subset Selector

To work properly, the design explorer requires a representative subset of application scenarios. The better the fitness prediction in the design explorer, the better the outcome of the scenario-based DSE is. Therefore, the subset selector is responsible for selecting a subset of scenarios. This subset selection is not trivial. First of all, there are a potentially large number of scenarios to pick from. This leads to a huge number of possible scenario subsets. On top of that, the scenario subset cannot be selected statically as the representativeness of the scenario subset is dependent of the current set of mappings. This set of mappings is only available at run time. Therefore, the scenario subset is selected dynamically.

At the end of the previous section, it was already explained that the design explorer communicates its current mapping population to the subset selector. This set of mappings can be used to *train* the scenario subset such that it is representative for the current population in the design explorer. As the population of the design explorer slowly changes over time, the subset will change accordingly. The overview of the scenario-based DSE (see Fig. 9.3) shows that the subset selector contains two threads of execution: the selector thread and the updater thread. Figure 9.7 shows these threads in more detail. The updater thread obtains the mapping individuals from the design explorer and updates the training set T_i of application mappings. This set of training mappings is used by the selector thread for selecting a representative subset of scenarios. The most representative subset of scenarios is exported to the design explorer.

In the remainder of this section, we provide a detailed overview of the subset selector. Before doing so, however, we will first describe the updater thread that is responsible for updating the trainer. Next, the metric used to judge the quality of the

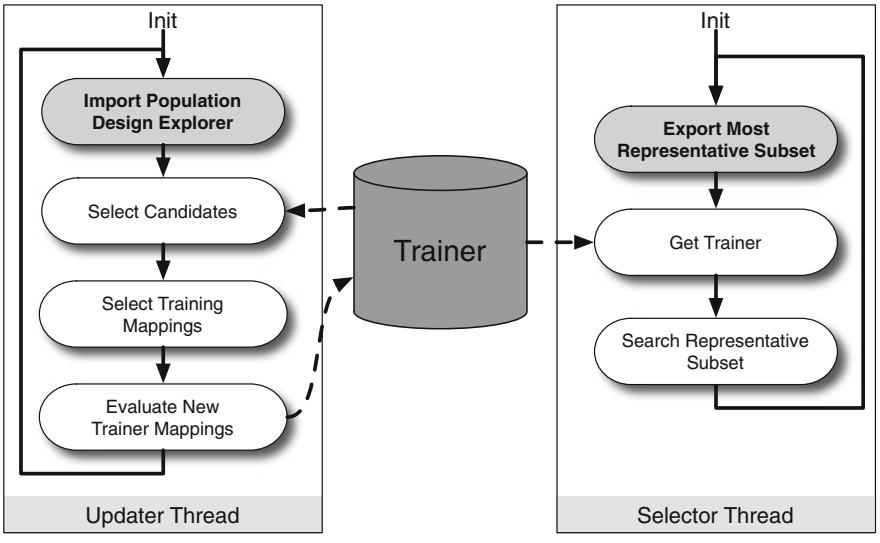


Fig. 9.7 The design of the subset selector

scenario subsets is described. The final subsection will show how the subset quality metric is used within the selector thread to select scenario subsets.

9.5.1 The Updater Thread

During the search of a representative subset of scenarios, it is crucial to have a set of training mappings T_i. Without a set of exhaustively evaluated mappings, one cannot judge if the predicted fitness of a scenario subset makes a correct prediction. As a training mapping is evaluated for all scenarios, it is relatively expensive to evaluate a mapping that needs to be added to the trainer. Therefore, it is important that the training mappings are selected carefully. Figure 9.7 illustrates the trainer update from T_i to T_{i+1}. The steps are as follows:

Import Population Design Explorer: To keep the training set T_i up to date with the mapping population in the design explorer, the current design explorer population g_j is imported.

Select Candidates: The current population is used to update the list of candidate mappings C_{i+1}:

$$\begin{aligned} &\underset{C_{i+1}}{\text{maximize}} \sum_{m \in C_{i+1}} \text{last\_gen}(m) \\ &\text{subject to (1) } C_{i+1} \subseteq C_i \cup g_j \\ &\qquad\qquad (2)\ C_{i+1} \cap T_i = \emptyset \\ &\qquad\qquad (3)\ |C_{i+1}| \leq \text{C\_SIZE} \end{aligned}$$

While updating the candidate mappings, there are three conditions. First, the new set of candidate mappings is the union of the previous set of candidate mappings and the population g_j that was just received from the design explorer. Secondly, condition (2) makes sure that all the candidate mappings are not yet in the trainer. Using these two conditions, the procedure selects a set of candidate mappings that is new to the trainer. Still, the first two conditions do not provide any control on the size of the set of candidate mappings. As the selection of training mappings involves computational overhead, the size of the set of candidate mappings must be limited as well. Therefore, condition (3) makes sure that the size of the set of candidate mappings is not larger than the predefined constant C_SIZE. As the optimization goal is to maximize the sum of the last generation that each of the training mappings was used (as returned by the function last_gen), the most recently used mappings will be kept in the set of candidate mappings (these have the highest value for last used generation). Additionally, the optimization of the total sum tries to get the number of candidate mappings as large as possible: the least recently used candidate mappings will be removed until the set of candidate mappings is smaller or equal to C_SIZE. In this way, the representative subset of scenarios can be optimized to predict the fitness of the current population of the design explorer.

Select Training Mappings: For each of the candidate mappings, the predicted fitness using the currently most representative subset of scenarios is obtained. For the candidate mappings that have just been imported from the design explorer, this should not require any new Sesame simulations (the population is quite likely evaluated using the same representative subset). For older candidate mappings, some computational overhead may be required for the partial reevaluation of the mapping fitness. Together with the mappings in the current trainer T_i, an estimated Pareto front $\tilde{P}_j$ is obtained.

The main goal of the representative subset is to correctly identify good mappings. Therefore, the trainer will focus on the mappings that are the closest to the Pareto front. Any mapping may have a fitness that is hard to predict (a mapping with a high quality or a mapping with a poor quality), but the scenario-based DSE only suffers from high-quality mappings that have an incorrectly predicted fitness. As long as both the real and predicted fitness of a mapping are bad, it does not really matter how bad the predicted fitness is. However, it is still an issue if the predicted quality of a mapping is poor, whereas the real quality is good. In this case, the mapping will not be added to the trainer. Although this is undesirable, without exhaustively evaluating the candidate mappings, these kinds of incorrect predictions cannot be detected. As the exhaustive evaluation is expensive, the gain in the trainer quality does not outweigh the additional computational overhead that is required to identify the high-quality mappings where the predicted mapping quality is low. Over time the predicted ordering of the mappings near the predicted Pareto front will be improved. Likely, this will also improve the prediction of other mapping individuals.

Therefore, the k new training mappings M_c are selected from the set of candidate mappings C_{i+1} by optimizing the distance to the estimated Pareto front:

$$\underset{M_c}{\text{minimize}} \sum_{m \in M_c} \min_{m_p \in \tilde{P}_j} \left(\overline{d} \left(\tilde{F}_{\tilde{S}_j}(m_p), \tilde{F}_{\tilde{S}_j}(m) \right) \right)$$

$$\text{subject to (1) } M_c \subseteq C_{i+1}$$

$$(2) |M_c| = \min(|C_{i+1}|, k)$$

The mappings are ordered on their normalized Euclidean distance to the closest mapping in the estimated Pareto front $\tilde{P}_j$. Here, the normalized Euclidean distance $\overline{d}$ between solutions x_1 and x_2 (with f being the fitness function and n the number of optimization objectives) is defined as:

$$\overline{f_i}(x) = \frac{f_i(x) - f_i^{\min}}{f_i^{\max}(x) - f_i^{\min}}$$

$$\overline{d}(x_1, x_2) = \sqrt{\sum_{i=1}^{n} (\overline{f_i}(x_1) - \overline{f_i}(x_2))^2}$$

The normalized distance translates all the objectives to a range between 0 and 1. For this purpose, the minimal ($f_i^{\min}$) and maximal value ($f_i^{\max}$) for the objective must be known. From the candidate mappings (condition 1), the k mappings are selected (condition 2) that are the closest to the estimated Pareto front.

Evaluate New Training Mappings: The mappings that are selected are exhaustively evaluated using Sesame. For this purpose, a separate pool of Sesame workers is used (just as the design explorer has a pool of Sesame workers). Once the real fitness is known, the mappings can be used to generate trainer T_{i+1} out of trainer T_i. Before the new training mappings are added, the trainer is truncated to fit the new training mappings. This is done in such a way that the trainer always contains real Pareto front P:

$$\underset{T_{i+1}}{\text{minimize}} \sum_{m \in T_{i+1}} \min_{m_p \in P} \left(\overline{d} \left(F(m_p), F(m) \right) \right)$$

$$\text{subject to (1) } T_{i+1} \subseteq T_i$$

$$\text{(2) } |T_{i+1}| = \min(|T_i|, \text{T\_SIZE} - |M_c|)$$

The truncated new trainer is a subset of the old trainer (condition 1), and it does not exceed the predefined trainer size. If trainer mappings must be discarded, the mappings that are the furthest from the real Pareto front are removed. This is done because of the second purpose of the trainer: at the end of the scenario-based DSE, it contains the best mappings that are found over time with their real fitness. Hence, we assume that the maximal trainer size is picked in such a way that it is significantly larger than the size of the Pareto front P. After truncation, the next trainer can be finalized: $T_{i+1} = T_{i+1} \cup M_c$.

9.5.2 Subset Quality Metric

Having a set of training mappings is not sufficient for judging the quality of the scenario subsets. To determine the actual quality of a subset of representative scenarios, we use the *misclassification rate* metric. The misclassification rate counts the number of ranks that is predicted incorrectly. Before we go into the definition of the misclassification rate, we first take a look into the Pareto ranking [31]. There are several approaches to rank individuals using the Pareto dominance relations, but in this chapter, we only focus on two of those: Boolean ranking and Goldberg's ranking (also called non-dominated sorting).

The ranking schemes are visualized in Fig. 9.8. Goldberg's ranking approach uses the dominance depth of the individuals. This is the same approach as the NSGA-II selector. Boolean ranking, on the other hand, follows a more simple approach: if the solution is non-dominated, the rank is one; otherwise the rank is two. As the design explorer uses an NSGA-II-based GA, it may be straightforward to use Goldberg's ranking scheme for the misclassification rate. The Boolean ranking,

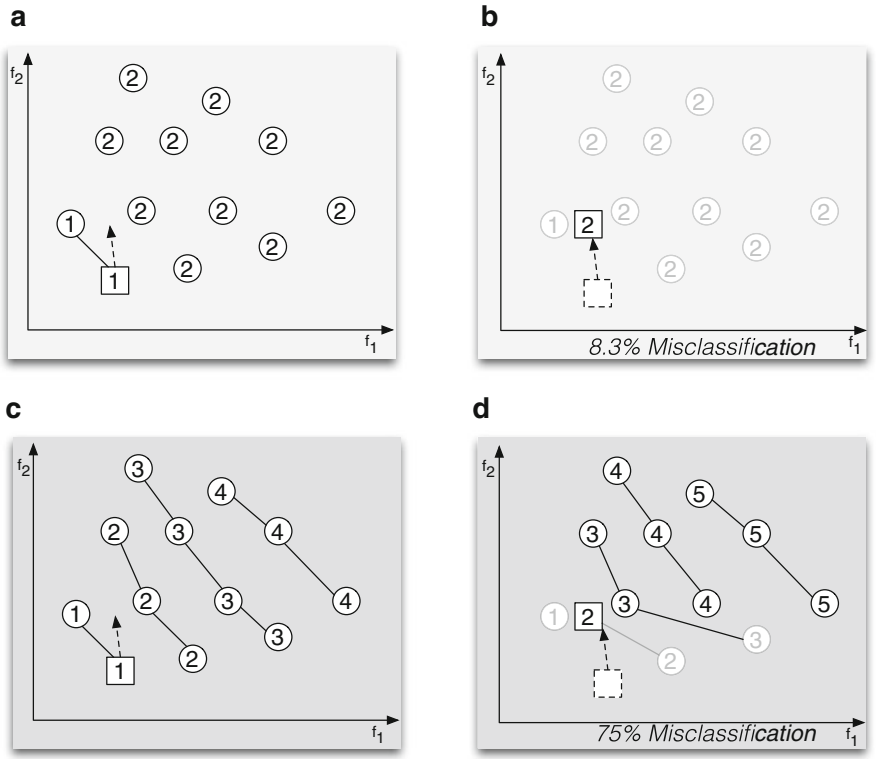


Fig. 9.8 A set of Pareto fronts showing the effect of a small error in the prediction (as shown with the dashed arrow) on the misclassification rate using the Boolean and Goldberg's ranking scheme. (**a**) Boolean ranking, real fitness. (**b**) Boolean ranking, predicted fitness. (**c**) Goldberg's ranking, real fitness. (**d**) Goldberg's ranking, predicted fitness

however, can be obtained more efficiently than the Goldberg's ranking. On top of that, the misclassification rate may be deceiving when Goldberg's ranking is used.

In Fig. 9.8, an example of such a deceiving case is given. The used trainer consists of 12 training mappings. Figure 9.8a, c show the exact fitness of the training mappings, whereas Fig. 9.8b, d show the predicted fitness of a specific scenario subset. This scenario subset provides a relatively good prediction: 11 out of the 12 training mappings are predicted correctly (the circular mappings). The incorrectly predicted training mapping (the square mapping) is slightly off as shown by a dashed arrow. Due to the incorrect prediction, the square mapping seems to be dominated by the leftmost training mapping. For both ranking schemes, the rank of the square mapping becomes ranked second instead of first. In case of the Boolean ranking, this is the only rank that is incorrect. For Goldberg's ranking, however, all the training mappings that are dominated by the square mapping are also incremented by one. As a result, the Goldberg's ranking has a misclassification rate of $\frac{3}{4}$, whereas the

Boolean ranking has a misclassification rate of $\frac{1}{12}$. Our example clearly shows that a high-quality mapping that is incorrectly ranked can affect all of its dominated solutions. However, as the main purpose of scenario-based DSE is that the Pareto front is predicted correctly, it is not a problem that the poor mappings are incorrectly ranked.

This is exactly what is determined with Boolean ranking. Each rank is based on the correct prediction of a non-dominated individual. A formal definition of the Boolean ranking is given in the following equation:

$$\mathrm{rel}_{F'}(m_1, m_2) := \begin{cases} 1 & F'(m_1) \text{ dominates } F'(m_2) \\ -1 & F'(m_2) \text{ dominates } F'(m_1) \\ 0 & else \end{cases} \tag{9.12}$$

$$\mathrm{rank}_{F'}(m, T) := \begin{cases} 2 & \exists m' \in T \ (\mathrm{rel}_{F'}(m', m) = 1) \\ 1 & else \end{cases} \tag{9.13}$$

Equation 9.12 formally defines the Pareto dominance between two mappings m_1 and m_2. The mappings are evaluated using fitness function F'. This can be the real fitness F but also the predicted fitness $\tilde{F}_{\tilde{S}}$. In this case, the scenario subset $\tilde{S}$ is used to predict the fitness of the mappings.

Based on the $\mathrm{rel}_{F'}$ function, the $\mathrm{rank}_{F'}$ is defined. This function ranks mapping m given trainer T (see Eq. 9.13). In case any of the mappings in the trainer dominates the mapping, the rank is equal to two. Otherwise, the mapping is Pareto optimal, and the rank is equal to one. Given the ranking function, the misclassification rate can be defined:

$$\mathrm{r}_{\mathrm{rank}}(\tilde{S}, T) := \frac{|\{m \in T : \mathrm{rank}_F(m, T) \neq \mathrm{rank}_{\tilde{F}_{\tilde{S}}}(m, T)\}|}{|T|} \tag{9.14}$$

The rate of misclassified Boolean ranks is too coarse grained to be used in isolation. In contrast to, e.g., Spearman's rank correlation [26], a lower misclassification rate is always better (the more non-dominated individuals that are correctly identified, the better). However, the probability of an equal misclassification rate is quite likely.

In this case, the *number of misclassified relations* is used as a tiebreaker. The number of misclassified relations can be defined quite straightforwardly:

$$\mathrm{r}_{\mathrm{rel}}(\tilde{S}, T) := \frac{|\{m_1, m_2 \in T : \mathrm{rel}_F(m_1, m_2) \neq \mathrm{rel}_{\tilde{F}_{\tilde{S}}}(m_1, m_2)\}|}{|T|^2} \tag{9.15}$$

By definition, when the number of misclassified relations is zero, the number of misclassified ranks is also zero. For the other cases, the number of misclassified

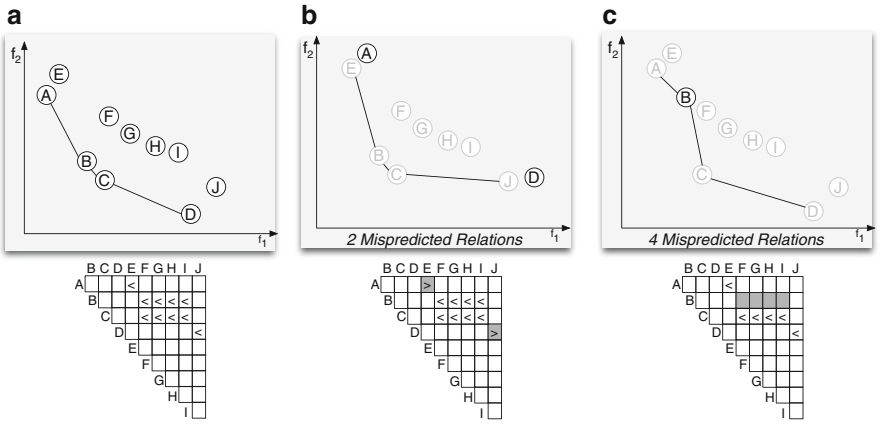


Fig. 9.9 A larger number of misclassified relations do not strictly correlate with the Pareto front quality. The Pareto front in (**c**) has more mispredicted relations than the front in (**b**), but the error ratio with respect to the real front is better. (**a**) Real. (**b**) 50% error ratio. (**c**) 0% error ratio

relations can suffer from the same problem as we showed with Goldberg's ranking. An example is shown in Fig. 9.9. Figure 9.9a shows the real Pareto front, where the fronts of Fig. 9.9b, c are obtained using a predicted fitness. The first prediction (Fig. 9.9b) only has two mispredicted relations (A $\leftrightarrow$ E and D $\leftrightarrow$ J), whereas the second prediction (Fig. 9.9c) has four mispredicted relations. Still, the Pareto front of the first prediction is only correct for 50% (E and J are not Pareto optimal). The second prediction, which is worse according to the number of misclassified relations, correctly identifies the Pareto front. As we are using the number misclassified relations as a subordinate metric, and not as a main metric, this is no problem in our case. Figure 9.9b has a misclassification rate of 20% that is worse than the misclassification rate of 0% in Fig. 9.9c.

9.5.3 The Selector Thread

The selector thread uses the subset quality metrics to select the representative subset of scenarios. More specifically, the goal of the selector thread is as follows:

$$\underset{\tilde{S}}{\text{minimize}}\ \mathrm{r}_{\text{rank}}\left(\tilde{S}, T\right) : \ \underset{\tilde{S}}{\text{minimize}}\ \mathrm{r}_{\text{rel}}\left(\tilde{S}, T\right) \tag{9.16}$$

As discussed in the previous subsection, the main goal is to optimize the quality of the predicted ranking. In the case of ties, the number of mispredicted relations will determine which of the scenario subsets is the best. Whenever a better representative subset is found, the subset is shared with the design explorer in order to improve its fitness prediction. The subset may be of any size, as long as it does not exceed a user-defined limit. This means that a smaller subset that has a better or equal

representativeness is preferable to a larger counterpart (the smaller the subset is, the faster the fitness prediction is).

This leaves us with the question of how to dynamically search for the representative subset of scenarios. In [28], it was shown that a random pick of scenarios does not result in a representative subset of scenarios. In the following subsections, we describe three different techniques for this searching process, using (1) a genetic algorithm, (2) a feature selection algorithm, and (3) a hybrid combination of both a genetic algorithm and feature selection.

9.5.3.1 GA-Based Search for Scenario Subset

Our first subset selection technique uses a genetic algorithm (GA) to select the representative scenario subsets. The GA of the subset selector is somewhat similar to the GA in the design explorer: a population of individuals is evolved over time in order to find the individual with the highest fitness. In order to describe the individual, a chromosome is used that enumerates the scenarios that are contained in the scenario subset. This chromosome, as illustrated in Fig. 9.10, is simply a sequence of integers that refer to scenarios from the scenario database. As the length of the chromosome is equal to the limit of the scenario subset size, the scenario subset can never become too large. Smaller scenario subset sizes are achieved in two ways: (1) scenarios may be used more than once within the same chromosome and (2) there is a special value for an unused slot in the scenario subset.

Scenario subsets can change size as an effect of the mutation or crossover that is applied during the search of a GA. The evolutionary process uses mutation and crossover to prepare individuals for the next generation of scenario subsets. Where the mutation replaces the scenarios in the subset one by one with other scenarios, the crossover partly exchanges the scenarios of two subsets. Only the successful modifications will make it to the next generation, leading to a gradual improvement of the representative subset of scenarios.

This approach has several benefits. First, the computational overhead is relatively small. Most time is spent in judging the quality of the scenario subsets and modifying the population of scenario subsets. Additionally, selecting scenario subsets for the next generation is relatively cheap. Apart from the low computational overhead, the search can also quickly move through the total space of scenario

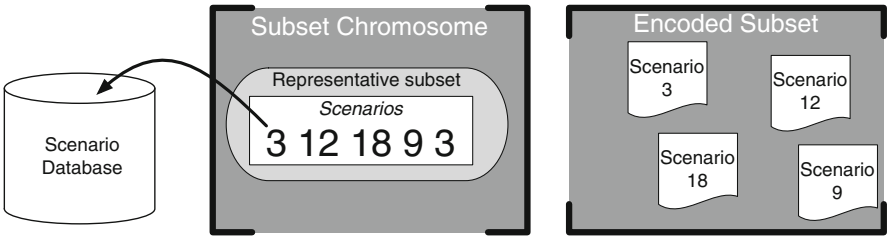


Fig. 9.10 Chromosome representation of a subset. Both the gene sequence is shown and the representative subset of scenarios that is encoded by the gene sequence

subsets. Due to crossover and mutation, the scenario subset can quickly change, and the alternatives can be evaluated and explored. This also means that local optima can easily be avoided. A local optimum is a solution that is clearly better than all closely related solutions. A small change of a local optimum does typically not result in a better subset, but the local optimum may still be much worse than the global optimal solution. As a genetic algorithm always has a small probability that the scenario subsets are changed significantly in the next generation, there is always a probability that the search will escape from the local optimum.

Unfortunately, this is also the downside of the approach. Just when the search comes close to the optimal solution, the mutation can quickly move the search into a completely different direction. Although the likeliness of this all depends on the choice of parameters such as mutation and crossover probability, it may be quite hard to pick the parameters in such a way that the search space is completely explored in the promising regions. Elitism in GAs assures that the points close to the local optimum will be retained as part of the population, but not that the neighborhood of each solution is carefully explored.

9.5.3.2 FS-Based Search for Scenario Subset

It would be better if the approach was less dependent on the choice of the parameters of the search algorithm. The Feature Selection (FS) technique has less parameters, and it basically performs a guided search that tries to improve a single scenario subset step by step. There are many different feature selection techniques, each giving a different trade-off between computational overhead and its quality. In fact, the feature selection techniques with the lowest computational overhead actually use a GA. In our case, we have chosen to use the dynamic sequential oscillating search [25] as, in general, it provides better classifiers (i.e., the scenario subset that classifies the non-dominated mapping individuals), with a moderate computational overhead.

Figure 9.11 illustrates the dynamic oscillating search. The most fundamental part of the algorithm is the up- and downswing. These swings are named according to their effect on the size of the scenario subset. Where the upswing will modify the subset by first adding a number of scenarios to the subset and then removing the same number of scenarios, the downswing will first remove scenarios before new scenarios are added again. This explains the name of the upswing and downswing:

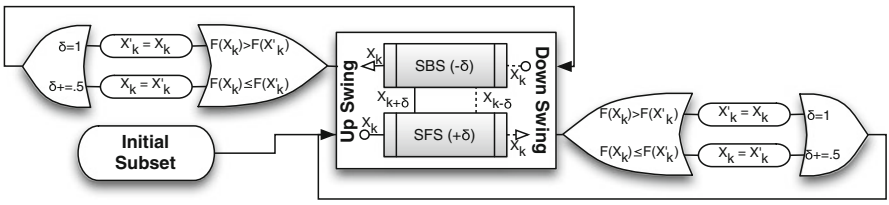


Fig. 9.11 An illustration of feature selection by means of the dynamic oscillation search procedure

in case of the upswing, the size of the subset swings upward, and for a downswing, it swings downward. For adding and removing scenarios, the *Sequential Forward Selection* (SFS) and *Sequential Backward Selection* (SBS) are used. These techniques will iteratively add or remove a scenario in the most optimal way. This means that SFS will increase the scenario subset size by iteratively adding the most optimal scenario. This most optimal scenario is determined by trying out all possible scenarios from the scenario database, and the scenario that results in the best scenario subset will be added to the larger subset. Similarly, the SBS will iteratively choose the optimal scenario to remove from the scenario subset. This means that all the scenarios that can be removed from the scenario subset are tried and the scenario removal that results in the best scenario subset will be applied.

As simple as it sounds, it makes the computational overhead of the swings largely dependent on the number of scenarios that are added or removed. The number of possibilities will grow linearly with respect with the number of scenarios that are added and removed during the swing. For each scenario that is added, all scenarios in the scenario database must be analyzed. Since this leads to a quick increase of computational overhead once the swings become larger, the size of the swing (or δ as used in Fig. 9.11) is initialized to one and slowly increased. During the search, the up- and downswing are alternated, and whenever both the up- and downswing do not result in a better scenario subset, the size of the swing is incremented by one. This can be seen in Fig. 9.11, at the cases where $F(X_k) \leq F(X'_k)$. The subset X'_k is the current best subset, and the subset X_k is the subset that is obtained after the up- or downswing. As a higher value for the function F means a better scenario subset, the case where $F(X_k) \leq F(X'_k)$ is an unsuccessful attempt to improve the scenario subset by a swing. Therefore, the currently best subset is restored, and the size of the swing is increased by 0.5. The value 0.5 is used to increment the swing by one after two unsuccessful swings: the number of scenarios that are added is the truncated integer value of δ. Of course, the swing can also be successful: in that case, the current best subset X'_k is updated, and the swing size is reset to one.

In a sense, the dynamic oscillating search is a kind of hill climbing technique. It oscillates the size of the scenario subset by exhaustively exploring all possibilities to change the scenario subset. Whenever a better subset is found, the current best representative subset is updated. Important to realize is that the current best subset can also be updated during a swing. As SFS and SBS analyze the quality of the scenario subset for each scenario that is added, it can be the case that a better representative subset is found during the swing. If the size of this subset is smaller than the maximal size, the currently most representative subset is updated and sent to the design explorer.

The FS method is more directed than the GA, and, therefore, it will only move closer to the optimal scenario subset. Unfortunately, this comes at a price: the FS is much more sensitive to local optima than the GA approach.

9.5.3.3 A Hybrid Approach for Searching Scenario Subsets

Ideally, we want to combine the strengths of the GA and the FS approaches. The hybrid approach (as shown in Fig. 9.12) tries to achieve this by alternating the GA

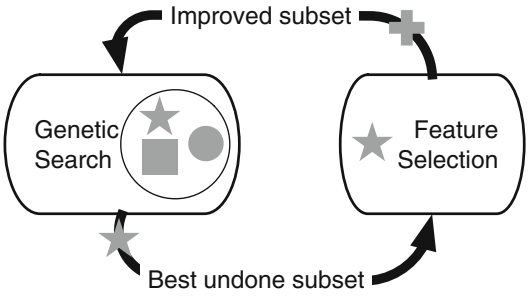


Fig. 9.12 The hybrid subset selection approach that alternates between a GA and a FS algorithm

and the FS methods. During the search for the representative subset, a GA will quickly prune the design space of potential scenario subsets, whereas the FS will thoroughly search the small neighborhood around the high-quality scenario subsets that are found by the GA. The tricky point is the moment of alternation. When one of the methods starts to converge, the other method should be activated.

At first sight, the feature selection may be interpreted as a custom variation operator for the GA, but this is absolutely not the case. Both the GA and the FS will keep state over time, and, thus, if the same subset is sent to the FS more than once, the oscillating search will be continued where it stopped in the previous invocation.

As the GA keeps a population of scenario subsets and the FS only works on a single scenario subset, it must be determined which scenario subset from the GA population is sent to the FS selection method. The most obvious method is to send the most representative subset from the GA to the FS. This can, however, not be done indefinitely. If the same subset is sent to the FS too often, the hybrid approach will again be susceptible for getting stuck in local optima as all the effort of the FS will be spent on the same subset. Therefore, the amount of effort spent by the FS to improve a single scenario subset is limited. If the FS has spent sufficient time on the same scenario subset (this time can be spread over multiple invocations of the FS), the subset is done, and it will not be sent to the FS anymore. So, the subset is only sent if it is"unfinished": the size of the swing in the oscillating search does not exceed a predefined maximal margin. This margin is chosen in such a way that the computational overhead of a single swing is still acceptable.

9.6 Related Work

In recent years, much research has been performed on high-level modeling and simulation for MPSoC performance evaluation as well as on GA-based DSE [5, 8]. However, the majority of the work in this area still evaluates and explores systems under a single, fixed application workload. Some research has been initiated on recognizing workload scenarios [7, 17] and making DSE scenario aware [16, 32]. In [7], for example, different single-application scenarios are used for DSE. Another type of scenario is the use-case. A use-case can be compared with what we call

inter-application scenarios, and consequently, a use-case describes which applications are able to run concurrently. Examples of frameworks utilizing use-cases for mapping multiple applications are MAMPS [14] and the work of Benini et al. [1]. MAMPS is a system-level synthesis tool for mapping multiple applications on a FPGA, whereas Benini et al. use logic programming to reconfigure an embedded system when a new use-case is detected. Another way of describing the use of multiple applications is a multimode multimedia terminal [9], in which the inter-application behavior is captured in a single, heterogeneous Model of Computation (MoC) combining dataflow MoCs and state machine MoCs.

9.7 Discussion

Scenario-based DSE efficiently explores the mapping of dynamic multi-application workloads on an MPSoC platform. Crucial for the efficiency of such mapping exploration is the subset selector that dynamically selects the fitness predictor for the design explorer. This fitness predictor is a subset of application scenarios that is used by the design explorer to quickly identify the non-dominated set of MPSoC mappings. In this chapter, we have given a detailed description of how the representativeness of a scenario subset can be calculated and which techniques can be used to select the fitness predictor (i.e., the subset of scenarios).

The three different fitness prediction techniques that were presented are (1) a genetic algorithm (GA), (2) a feature selection (FS) algorithm, and (3) a hybrid method (HYB) combining the two aforementioned approaches. A genetic algorithm is capable of quickly exploring the space of potential scenario subsets, but due to its stochastic nature, it is susceptible to missing the optimal scenario subsets. This is not the case with the feature selection algorithm as it more systematically explores the local neighborhood of a scenario subset. Unfortunately, this approach is relatively slow and can suffer from local optima. The solution is to combine these approaches in the hybrid approach, leading to a fitness prediction technique that can quickly prune the design space, can thoroughly search the local neighborhood of scenario subsets, and is less susceptible to local optima.

To give a feeling of the performance of the three different fitness prediction techniques, Fig. 9.13 shows the results of a scenario-based DSE experiment in which the three techniques are compared for three different subset sizes (1, 4, and 8% of the total number of application scenarios). In this experiment, the mapping of ten applications with a total of 58 processes and 75 communication channels is explored. The multi-application workload consists of 4607 different application scenarios in total. The target platform is a heterogeneous MPSoC with four general-purpose processors, two ASIPs and two ASICs, all connected using a crossbar network. In this experiment, we have measured the required exploration time for the scenario-based DSE to identify a satisfying mapping. After all, the faster the DSE can provide results that match the requirement of the user, the better it is. For this purpose, a DSE of 100 min is performed for all three subset selector approaches. The results have been averaged over nine runs. To determine the efficiency of the

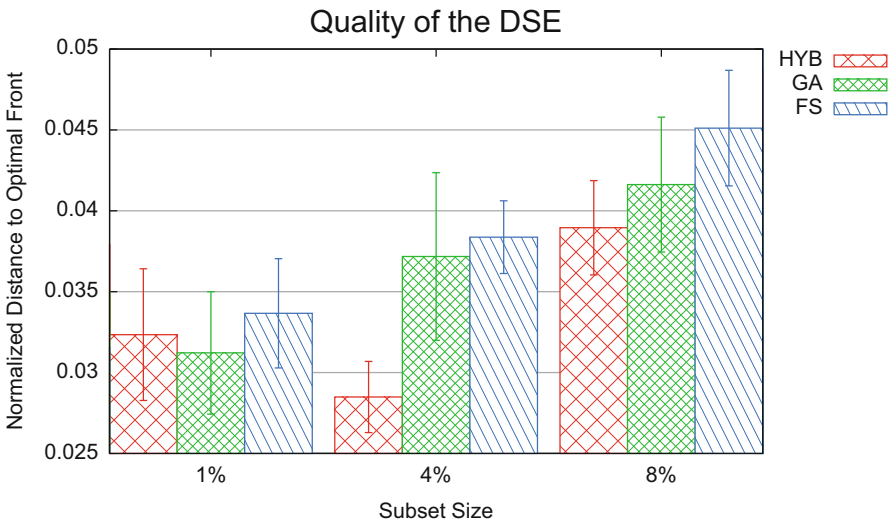


Fig. 9.13 Quality of the DSE for the different subset selection approaches. The quality is determined based on the distance between the estimated Pareto front and the optimal front

multi-objective DSE, we obtain the distance of the estimated Pareto front (execution time versus energy consumption of mapping solutions) to the optimal Pareto front. For this purpose, we normalized execution time and energy consumption to a range from 0 to 1. As the optimal Pareto front is not exactly known since the design space is too large to exhaustively search it, we have used the combined Pareto front of all our experiments for this.

When increasing the subset size, two effects will occur: (1) the larger the subset, the more accurate the fitness prediction in the design explorer is and (2) the larger the subset, the longer it takes to obtain the fitness of a single mapping causing a slower convergence of the search. This can be seen in Fig. 9.13. The GA and the FS subset selection have worse results when the subset becomes larger (the smaller the distance, the better). The hybrid selector, however, shows a somewhat different effect. With a subset size of 4%, it is able to benefit from a subset with a higher accuracy. The slower convergence only starts to effect the efficiency for the 8% subset. Comparing the different methods, the hybrid method shows the best results. The only exception is for the 1% subset. In this case, the GA is still able to search the smaller design space of possible subsets. Still, the result of the hybrid method at 4% is better than the result of the GA at 1%. With the larger subset sizes, the hybrid method can exploit both the benefits of the feature selection and the genetic algorithm.

For more extensive experimental evaluations of scenario-based DSE, and the different fitness prediction techniques in scenario-based DSE in particular, we refer the interested reader to [28, 30]. These studies compare scenario-based DSE with regular DSE in the context of multi-application workloads. Moreover, they

scrutinize the quality of the mapping solutions obtained by the different variants of scenario-based DSE as well as the efficiency with which these solutions are obtained.

The scenario-based DSE presented in this chapter aims at providing a *static mapping* of a multi-application workload onto an MPSoC. Evidently, the application dynamism as captured by application scenarios can of course also be exploited at run time to *dynamically optimize* the embedded system according to the application workload at hand. For example, in [20–23] as well as in ▸ Chap. 10, "Design Space Exploration and Run-Time Adaptation for Multicore Resource Management Under Performance and Power Constraints", various approaches are proposed for adaptive MPSoC systems that allow for such dynamic system optimization. These methods typically consist of two phases: A design-time stage performs DSE to find an optimal mapping for each application. At run time, the occurrence of different application scenarios is detected, after which the system can be reconfigured by dynamically adapting the application mapping. This could, e.g., be done by merging the pre-optimized mappings of each separate, active application in the detected application scenario to form a first-order mapping for the entire scenario. Subsequently, this first-order mapping can then be further optimized by using run-time mapping optimization heuristics [22].

References

1. Benini L, Bertozzi D, Milano M (2008) Resource management policy handling multiple use-cases in MPSoC platforms using constraint programming. In: Logic programming, Udine. Lecture notes in computer science, vol 5366, pp 470–484
2. Coello CAC, Lamont GB, Veldhuizen DA (2007) Alternative metaheuristics. In: Coello Coello CA, Lamont GB, Van Veldhuizen DA (eds) Evolutionary algorithms for solving multi-objective problems. Genetic and evolutionary computation, 2nd edn. Springer, New York
3. Coffland JE, Pimentel AD (2003) A software framework for efficient system-level performance evaluation of embedded systems. In: Proceedings of the SAC 2003, pp 666–671
4. Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evol Comput 6(2):182–197
5. Erbas C, Cerav-Erbas S, Pimentel AD (2006) Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. IEEE Trans Evol Comput 10(3):358–374
6. Gerstlauer A, Haubelt C, Pimentel A, Stefanov T, Gajski D, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(10):1517–1530
7. Gheorghita SV et al (2009) System-scenario-based design of dynamic embedded systems. ACM Trans Des Autom Electron Syst 14(1):1–45
8. Gries M (2004) Methods for evaluating and covering the design space during early design development. Integr VSLI J 38(2):131–183
9. Ha S, Lee C, Yi Y, Kwon S, Joo YP (2006) Hardware-software codesign of multimedia embedded systems: the peace approach. In: Proceedings of the IEEE international conference on embedded and real-time computing systems and applications, pp 207–214
10. Jia Z, Bautista T, Nunez A, Pimentel A, Thompson M (2013) A system-level infrastructure for multidimensional MP-SoC design space co-exploration. ACM Trans Embed Comput Syst 13(1s):27:1–27:26

11. Kahn G (1974) The semantics of a simple language for parallel programming. In: Proceedings of the IFIP congress 74. North-Holland Publishing Co.
12. Keutzer K, Newton A, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543
13. Kienhuis B, Deprettere EF, van der Wolf P, Vissers KA (2002) A methodology to design programmable embedded systems: the Y-chart approach. In: Embedded processor design challenges. LNCS, vol 2268. Springer, Berlin/New York, pp 18–37
14. Kumar A, Fernando S, Ha Y, Mesman B, Corporaal H: Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. ACM Trans Des Autom Electron Syst 13(3):1–27 (2008)
15. Martin G (2006) Overview of the MPSoC design challenge. In: Proceedings of the design automation conference (DAC'06), pp 274–279
16. Palermo G, Silvano C, Zaccaria V: Robust optimization of SoC architectures: a multi-scenario approach. In: Proceedings of the IEEE workshop on embedded systems for real-time multimedia (2008)
17. Paul JM, Thomas DE, Bobrek A (2006) Scenario-oriented design for single-chip heterogeneous multiprocessors. IEEE Trans Very Large Scale Integr Syst 14(8):868–880
18. Pimentel A, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112
19. Piscitelli R, Pimentel A (2012) Design space pruning through hybrid analysis in system-level design space exploration. In: Proceedings of DATE'12, pp 781–786
20. Quan W, Pimentel AD (2013) An iterative multi-application mapping algorithm for heterogeneous MPSoCs. In: Proceedings of ESTIMedia'13, pp 115–124
21. Quan W, Pimentel AD (2013) A scenario-based run-time task mapping algorithm for MPSoCs. In: Proceedings of DAC'13, pp 131:1–131:6
22. Quan W, Pimentel AD (2015) A hybrid task mapping algorithm for heterogeneous MPSoCs. ACM Trans Embed Comput Syst 14(1):14:1–14:25
23. Schor L, Bacivarov I, Rai D, Yang H, Kang SH, Thiele L (2012) Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Proceedings of CASES'12, pp 71–80
24. Singh AK, Shafique M, Kumar A, Henkel J (2013) Mapping on multi/many-core systems: survey of current and emerging trends. In: Proceedings of DAC'13, pp 1:1–1:10
25. Somol P, Novovicova J, Grim J, Pudil P (2008) Dynamic oscillating search algorithm for feature selection. In: Proceedings of the international conference on pattern recognition (ICPR 2008), pp 1–4
26. Spearman C (1904) The proof and measurement of association between two things. Am J Psychol 15(1):72–101
27. van Stralen P (2014) Applications of scenarios in early embedded system design space exploration. PhD thesis, Informatics Institute, University of Amsterdam
28. van Stralen P, Pimentel AD (2010) Scenario-based design space exploration of MPSoCs. In: Proceedings of IEEE international conference on computer design (ICCD'10)
29. van Stralen P, Pimentel AD (2010) A trace-based scenario database for high-level simulation of multimedia mp-socs. In: Proceedings of SAMOS'10
30. van Stralen P, Pimentel AD (2013) Fitness prediction techniques for scenario-based design space exploration. IEEE Trans Comput-Aided Des Integr Circuits Syst 32(8):1240–1253
31. van Veldhuizen DA, Lamont GB (2000) Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. Evol Comput 8(2):125–147
32. Zaccaria V, Palermo G, Castro F, Silvano C, Mariani G (2010) Multicube explorer: an open source framework for design space exploration of chip multi-processors. In: Proceedings of the international conference on architecture of computing systems (ARCS), pp 1–7

Design Space Exploration and Run-Time Adaptation for Multicore Resource Management Under Performance and Power Constraints

10

Santiago Pagani, Muhammad Shafique, and Jörg Henkel

Abstract

This chapter focuses on resource management techniques for performance or energy optimization in multi-/many-core systems. First, it gives a comprehensive overview about resource management in a broad perspective. Secondly, it discusses the possible optimization goals and constraints of resource management techniques: computational performance, power consumption, energy consumption, and temperature. Finally, it details the state-of-the-art techniques on resource management for performance optimization under power and thermal constraints, as well as for energy optimization under performance constraints.

Acronyms

DPM	Dynamic Power Management
DSE	Design Space Exploration
DSP	Digital Signal Processor
DTM	Dynamic Thermal Management
DVFS	Dynamic Voltage and Frequency Scaling
EOH	Extremal Optimization meta-Heuristic
EWFD	Equally-Worst-Fit-Decreasing
GIPS	Giga-Instruction Per Second
GPP	General-Purpose Processor
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
IPS	Instruction Per Second
ISA	Instruction-Set Architecture
ITRS	International Technology Roadmap for Semiconductors
LTF	Largest Task First

S. Pagani (✉) • M. Shafique • J. Henkel
Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
e-mail: pagani@kit.edu; shafique@kit.edu; henkel@kit.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_11

MPSoC	Multi-Processor System-on-Chip
NoC	Network-on-Chip
QoS	Quality of Service
SCC	Single Chip Cloud computer
SFA	Single Frequency Approximation
SoC	System-on-Chip
TDP	Thermal Design Power
TLP	Thread-Level Parallelism
TSP	Thermal Safe Power

Contents

10.1 Introduction

In the past decade, single-core processors have reached a practical upper limit with respect to their maximum operational frequency, mostly due to power dissipation. This has motivated chip manufacturers to shift their focus toward designing processors with multiple cores which operate at lower frequencies than their single-core counterparts, such that they can potentially achieve the same computational performance while consuming less power. Furthermore, computational performance demands of modern applications have substantially increased and can no longer be satisfied only by increasing the frequency of a single-core processor or by customizing such a processor. In other words, modern computing systems require processors with multiple cores in the same chip (expected to increase in number every year, as shown in Fig. 10.1), which can efficiently communicate with each other and provide increased parallelism. The main idea is therefore to consider an application as a group of many small tasks, such that these tasks can be executed in parallel on multiple cores and thus meet the increased performance demands [1].

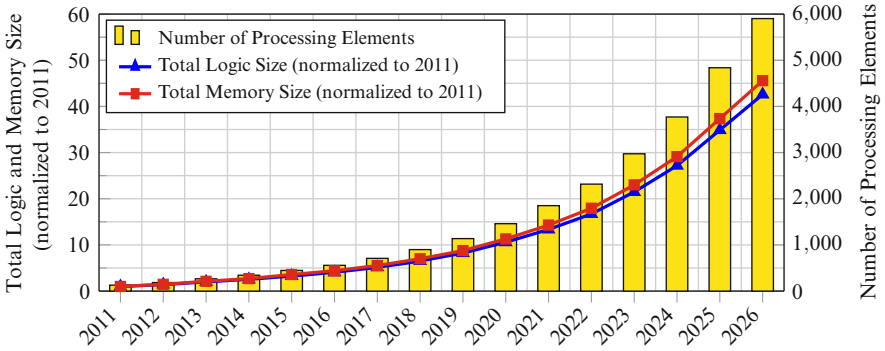


Fig. 10.1 System-on-Chip (SoC) consumer portable design complexity trends [24]. The figure shows the expected total logic and memory sizes (normalized to 2011), as well as the expected number of processing elements to be found in future SoCs, as predicted by ITRS in 2001

Aside for motivating the use of multi-core processors, the continuous increasing performance demands and power budget constraints have led to the emergence of heterogeneous multi-core processors composed by more than one type of core, where each core type has different performance and power characteristics [62]. The distinct architectural features of the different types of cores can be potentially exploited in order to meet both the functional and nonfunctional requirements (e.g., computational performance, power consumption, temperature, etc.). This makes heterogeneous multi-core and many-core systems a very promising alternative over their homogeneous counterpart, where an application may witness large improvements in power and/or performance when mapped to a suitable type of core, as also discussed in ▸ Chap. 8, "Architecture and Cross-Layer Design Space Exploration".

Furthermore, as nanotechnology evolves, it is expected that in upcoming years thousands of cores will be integrated on the same chip [24]. Such a large number of cores need a Network-on-Chip (NoC) based on an efficient and scalable interconnection infrastructure for core-to-core communication (as also discussed in ▸ Chap. 15, "Network-on-Chip Design"). To efficiently manage these processors, we need sophisticated *resource and power management* techniques, which can be categorized as *"centralized or distributed"* (with respect to their view of the system and information exchange during the management decision), and as *"design time or run time"* (with respect to the time of the decision making algorithms). Resource and power management techniques are responsible for mapping threads/tasks to specific resources on the chip (i.e., cores of different types, accelerators, remote servers, etc.), migrating threads/tasks at run time, managing the power modes of the resources (active, clock gated, sleep, power gated, etc.), selecting the voltage and frequency levels of the resources, etc. Considering all these management options leads to a very large design space from which to choose. For example, when executing N threads/tasks on a system with M cores, in which each core can run at F different voltage/frequency levels, there are $M^N + F^M$ possible mapping

and voltage/frequency combinations. According to the desired goals and given constraints, the appropriate combination can be therefore selected applying Design Space Exploration (DSE). Furthermore, most run-time techniques (and also several design-time techniques) considerably limit the design space in order to reduce the execution time of the management decisions.

10.1.1 Centralized and Distributed Techniques

Centralized techniques assume to have (or be able to obtain) a global view of the system. Namely, a centralized resource management technique would require to know what applications are being executed throughout the chip, in how many threads, mapped to which specific cores, executing at what voltage and frequency level, their power consumption, etc. With this information, a centralized technique can potentially arrive to very efficient resource management decisions. However, if such detailed information is not known by the centralized manager a priori (most likely scenario in real systems), then it needs to be transmitted through the corresponding communication infrastructure, requiring high communication bandwidths. This is the main reason why centralized techniques are generally not scalable, where scalability is defined as the ability of a technique to remain within feasible computation and communication constraints when the problem size grows (i.e., when we have an increasing number of cores on a processor). Therefore, *centralized techniques are well suited for processors with a limited number of cores*.

In practice however, a more realistic assumption, in terms of latency and communication bandwidth, is to assume that every core is restricted to a local view of the system where only information of the immediate surrounding neighborhood is known at any given time. This can be easily achieved by having cores periodically communicate with their neighbors, keeping the communication load distributed throughout the chip. In this way, the resource management decisions are made in a **distributed** fashion, where small groups of cores conduct local optimization, maintaining scalability. This local optimization is managed differently depending on the considered distributed technique. For example, each core could manage itself individually after exchanging information with other cores, or contrarily, some cores could act as local managers and make decisions that affect a small group of neighboring cores. In either case, the challenge for distributed techniques is to make high-quality resource management decisions with respect to the chosen optimization goal with such limited local information.

In summary, mainly due to scalability issues, distributed techniques are better suited for large multi-/many-core systems than centralized techniques. Specifically, in distributed systems:

(*continued*)

- The communication load is balanced throughout the processor, avoiding communication bottlenecks in the central manager.
- The computational complexity of the each distributed (local) resource management decision is much smaller than the complexity of a centralized decision, and therefore the problem size is decoupled from the total number of cores in the chip.

10.1.2 Design-Time Decisions and Run-Time Adaptations

Resource management and power management methodologies based on design-time decisions require to have advance knowledge of the applications to execute and their requirements, as well as an estimation of their characteristics (execution time, power consumption, etc.). Naturally, they have a global view of the system and are therefore generally implemented in a centralized manner. Furthermore, the execution time of design-time optimization algorithms is not of major importance, as long as it remains below reasonable limits (e.g., in terms of minutes or hours and not months or years). On the other hand, methodologies based on run-time adaptations only have information of the current application queue (i.e., list of applications ready for execution) and requirements. Moreover, the execution time of run-time optimization algorithms (also known as on-the-fly algorithms) contributes to the overall application execution time, and therefore, it is vital that they have a short execution time. Given that having a global knowledge of the system can potentially consume a high communication bandwidth and also requires considerable time, run-time methodologies are usually restricted to a local view of the system where only information of the immediate surrounding neighborhood may be available and are therefore generally implemented in a distributed manner.

In more detail, methodologies based on **design-time decisions**:

- Have a global view of the system.
- Do not have stringent timing constraints and therefore can involve complex dynamic programming algorithms [25, 44], integer linear programming [26], time-consuming heuristics (e.g., simulated annealing [33, 40] or genetic algorithms [11]), or a large DSE.
- Can generally result in higher quality decisions than run-time methodologies.
- Require previous knowledge (or predictions) of the system behavior, which is not always feasible to obtain a priori.
- Cannot adapt to changes in the system behavior, for example, the insertion of a new application unknown at design time.

Contrarily, methodologies based on **run-time (or on-the-fly) adaptations**:

- Only have information of the current application queue and requirements.
- Generally, restricted to a local view of the system having only information of the immediate surrounding neighborhood.
- Optimization decisions require a short execution time.
- Trade quality in order to reduce the communication bandwidth and the execution time.
- Generally implemented using (probably distributed) lightweight heuristics.
- Can adapt to changes in the system behavior, for example, to take advantage of dynamic workload variations of the applications (as also discussed in ▸ Chap. 9, "Scenario-Based Design Space Exploration", e.g., early completions, performance requirement changes, etc.) or to execute new applications unknown at design time (even after the delivery of the system to the end-user).
- Can adapt to hardware changes after the production of a System-on-Chip (SoC), for example, permanent hardware failures or core degradations due to aging effects.

Finally, aside from design-time or run-time algorithms, there exist **hybrid** methodologies which partially rely on results previously analyzed at design time and also on run-time adaptations [50, 56, 59, 60, 63, 68]. Namely, based on design-time analysis of the applications (stored on the system for a specific platform), lightweight heuristics can make very efficient run-time decisions that adapt to the system behavior (current applications on the ready queue, available system resources, desired performance, etc.). Such techniques still suffer from some of the downsides of design-time methods, for example, knowing all potential applications to be executed at design time such that they can be properly analyzed on the desired platform. Nevertheless, they can generally result in better resource management decisions than design-time algorithms (as they can adapt to the current system behavior) and than on-the-fly heuristics (as they can make more informed decisions or require less time to evaluate certain alternatives).

10.1.3 Parallel Applications

For an application to be executed in a multi-/many-core system, the application has to be parallelized (or partitioned) into multiple threads/tasks that can be concurrently executed on different cores. Although far from a solved problem, there exist some state-of-the-art application parallelization tools [7, 34] that can be used to take care of the task partitioning and manual analysis, involving finding a set of tasks, adding the required synchronization and inter-task communication to the corresponding tasks, management of the memory hierarchy, verifying the parallelized code in order to ensure a correct functionality [35], etc.

A task binding process is also required for the case of heterogeneous platforms, such that the system can know which tasks can be mapped to which type of cores

and with what cost [61]. Namely, depending on the underlying architecture, it is possible that not all tasks can be mapped to all core types, for example, a task requiring a floating point unit may not be mapped to a core that does not have such a unit. Moreover, the binding process must analyze the implementation costs (e.g., performance, power consumption, and resource utilization) of every task on all the types of cores that support each task, such as a General-Purpose Processor (GPP), a Digital Signal Processor (DSP), or some reconfigurable hardware.

10.2 Optimization Goals and Constraints

There are several possible optimization goals and constraints for resource management techniques: computational performance, power consumption, energy consumption, and temperature. For example, a system could choose to maximize the computational performance under a power or temperature constraint, while another system would prefer to minimize energy consumption under performance constraints.

10.2.1 Computational Performance

In few words, computational performance refers to how fast and efficiently can the system execute a given set of applications. It can be measured in many different ways, for example, application's execution time, throughput, Instructions Per Cycle (IPC), Instruction Per Second (IPS), normalized speed-up factor with respect to a known reference, etc. Generally, IPC and IPS are not well-suited metrics to use in heterogeneous systems, as different types of cores might have different Instruction-Set Architectures (ISAs) or require a different number of instructions to finish the same amount of work.

> Maximizing the *overall system performance* (generic term which can, e.g., refer to maximizing the summation of the weighted throughput of all applications, minimize the longest execution time among all applications, etc.) is generally the most commonly pursued optimization goal. Nevertheless, for applications with hard real-time deadlines, meeting the deadlines can be formulated as satisfying certain performance requirements, and therefore for such cases performance is considered as a constraint.

The execution time of an application (or the resulting performance of the application) will depend on how the application is executed, for example, in how many threads the application is parallelized in, the types of cores to which the application is mapped to, the execution frequency, the utilization of shared resources

(caches, NoC, etc.) by other applications, etc. The application's characteristics also play a major role in its execution time, for example, its Instruction-Level Parallelism (ILP) or its Thread-Level Parallelism (TLP) have a direct impact in how well an application scales with respect to frequency and the number of threads, respectively. Applications with high ILP generally scale well with increasing frequencies, while applications with high TLP generally scale better when parallelized in many threads. For example, Fig. 10.2 shows the execution time and speed-up factors of three applications from the PARSEC benchmark suite [3] with respect to the frequency when executing a single thread. Similarly, Fig. 10.3 shows the execution time and speed-up factors of the same application with respect to the number of parallel threads when executing at 2 GHz. From the figures, we can observe that the impact of the frequency and the number of parallel threads on the speed-up factors are entirely application dependent and that the application's performance will eventually stop scaling properly after a certain number of threads, known as the parallelism wall.

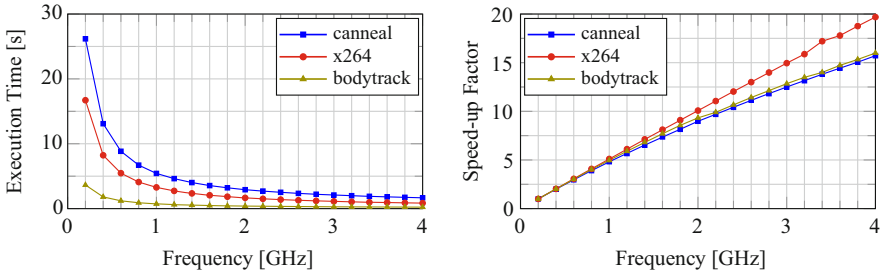


Fig. 10.2 Execution time and speed-up factors with respect to frequency based on simulations conducted on gem5 [4] for three applications from the PARSEC benchmark suite [3] executing a single thread on an *out-of-order* Alpha 21264 core. The speed-up factors are normalized to the execution time of each application running at 0.2 GHz

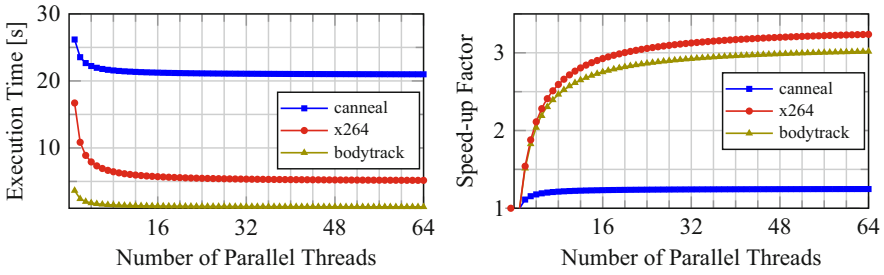


Fig. 10.3 Execution time and speed-up factors based on simulations conducted on gem5 [4] and Amdahl's law for three applications from the PARSEC benchmark suite [3] executing at 2 GHz on an *out-of-order* Alpha 21264 core. The speed-up factors are normalized to the execution time of each application running a single thread

10.2.2 Power and Energy Consumption

Power consumption is in nature an instantaneous metric which changes through time. Particularly, a core executing a certain thread of an application will consume different amounts of power at different time instants and application phases. For example, Fig. 10.4 illustrates the power consumption results of simulations conducted with Sniper [5] and McPAT [31], for a PARSEC bodytrack application executing four parallel threads on a quad-core Intel Nehalem cluster running at 2.2 GHz. The power consumption values observed on a specific core at a given point in time depend on several parameters, e.g., the underlying architecture of the core, the technology scaling generation, the mode of execution of the core (e.g., active, idle, or in a low-power mode), the selected voltage/frequency levels for execution, the temperature on the core (for leakage power thermal dependency), the application phase begin executed, etc.

Generally, as detailed in [17], the power consumption of a CMOS core can be modeled as formulated in Equation (10.1):

$$P = \alpha \cdot C_{\text{eff}}^{\text{app}} \cdot V_{\text{dd}}^2 \cdot f + V_{\text{dd}} \cdot I_{\text{leak}}\left(V_{\text{dd}}, T\right) + P_{\text{ind}} \tag{10.1}$$

where α represents the activity factor (or utilization) of the core, $C_{\text{eff}}^{\text{app}}$ represents the effective switching capacitance of a given application, V_{dd} represents the supply

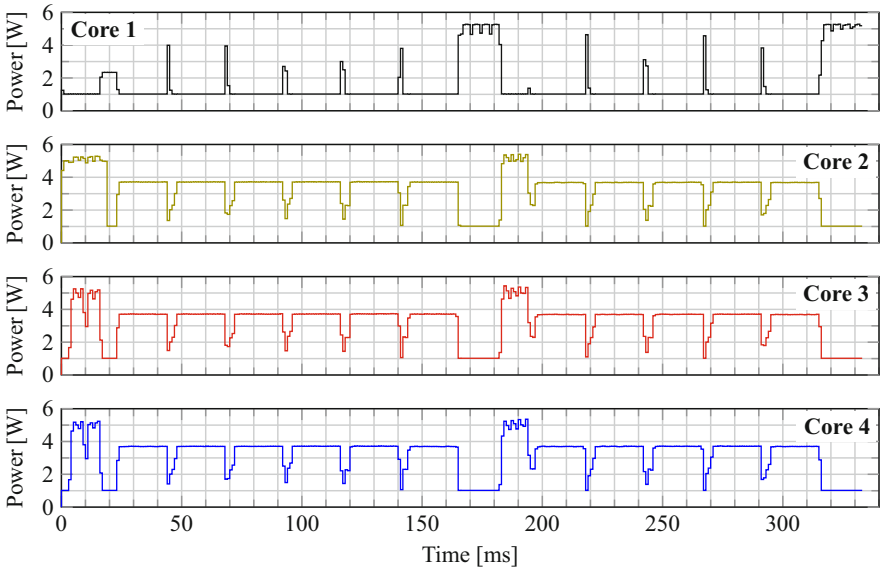


Fig. 10.4 Power consumption results of simulations conducted using Sniper [5] and McPAT [31], for a PARSEC bodytrack application with *simmedium* input, executing four parallel threads at 2.2 GHz on a quad-core Intel Nehalem cluster

voltage, f represents the execution frequency, I_{leak} represents the leakage current (which depends on the supply voltage and the core's temperature T), and P_{ind} represents the independent power consumption (attributed to keeping the core in execution mode, i.e., the voltage-/frequency-independent component of the active power consumption). Moreover, in Equation (10.1), $\alpha \cdot C_{\text{eff}}^{\text{app}} \cdot V_{\text{dd}}^2 \cdot f$ represents the dynamic power consumption, while $V_{\text{dd}} \cdot I_{\text{leak}}(V_{\text{dd}}, T)$ represents the leakage power consumption. Hence, if a core is clock gated, it still consumes leakage and indirect power. On the other hand, cores can also be set to some low-power mode (e.g., sleep or power-gated), each mode with an associated power consumption and different latencies for entering and leaving each low-power mode.

With respect to the voltage and frequency of the core, in order to stably support a specific frequency, the supply voltage of the core has to be adjusted above a minimum value. This minimum voltage value is frequency dependent, and higher frequencies require a higher minimum voltages. Furthermore, as shown by Pinckney et al. [49], the relation between the frequency and the corresponding minimum voltage can be modeled according to Equation (10.2):

$$f = k \cdot \frac{(V_{\text{dd}} - V_{\text{th}})^2}{V_{\text{dd}}} \tag{10.2}$$

where V_{th} is the threshold voltage and k is a fitting factor. Expressed in other words, the physical meaning of Equation (10.2) is that for a given supply voltage, there is a maximum stable frequency at which a core can be executed, and running at lower frequencies is stable but power/energy inefficient. Therefore, if the system runs at the corresponding power-/energy-efficient voltage and frequency pairs, we can derive a linear relationship between voltage and frequency and thus arrive at a *cubic* relation between the frequency and the dynamic power consumption. Figure 10.5 uses Equation (10.2) to model the minimum voltages necessary for stable execution on a 28 nm x86-64 microprocessor [14], and Fig. 10.6 shows how the power model from Equation (10.1) fits average power consumption results from McPAT [31] simulations for an x264 application from the PARSEC benchmark suite [3].

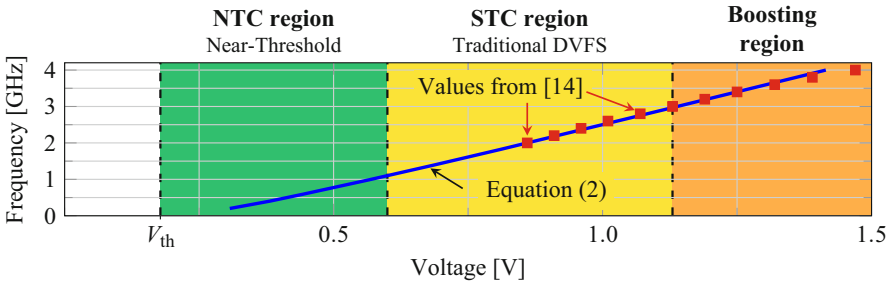


Fig. 10.5 Frequency and voltage relation modeled with Equation (10.2) for the *experimental results of a* 28 *nm x86-64 microprocessor* developed in [14]

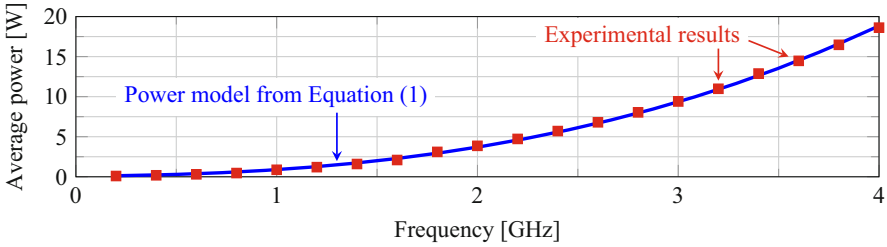


Fig. 10.6 Experimental results for a 22 nm *out-of-order* Alpha 21264 core, based on our simulations conducted on gem5 [4] and McPAT [31] for an x264 application from the PARSEC benchmark suite [3] running a single thread, and the derived power model from Equation (10.1)

a

Voltage	Frequency
0.73 V	301.48 MHz
0.75 V	368.82 MHz
0.85 V	569.45 MHz
0.94 V	742.96 MHz
1.04 V	908.92 MHz
1.14 V	1077.11 MHz
1.23 V	1223.37 MHz
1.32 V	1303.79 MHz

b

Voltage	Frequency
0.70 V	25.38 W
0.80 V	37.26 W
0.91 V	50.76 W
1.00 V	70.73 W
1.05 V	91.25 W
1.10 V	110.15 W
1.14 V	125.27 W
1.21 V	161.99 W
1.28 V	201.40 W

c

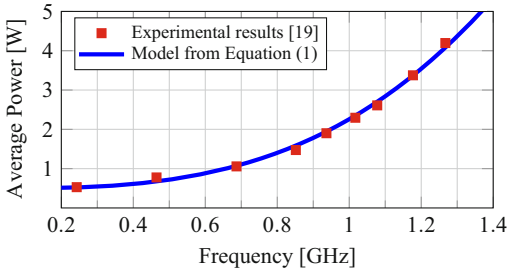


Fig. 10.7 Experimental results for the 48-core system developed in [19] and the power model from Equation (10.1). (**a**) Frequency vs. voltage [19] (**b**) Power vs. voltage [19] (**c**) Power model for a single core

Similarly, we can also use Equation (10.1) to model the experimental results from a research paper on Intel's Single Chip Cloud computer (SCC) [19], which developed a many-core system that integrates 48 cores. The work in [19], presents a relationship between the minimum voltages necessary for stable execution when running the cores at different frequencies, as well as average power consumption values for the entire chip when executing computational intensive applications running at the maximum allowed frequency for a given voltage, and these results are summarize in Fig. 10.7a, b. Therefore, we can fit the power model from Equation (10.1) based on the values of these two tables, such that the power consumption on individual cores can be modeled as illustrated in Fig. 10.7c.

Energy is the integration of power over time, and thus, when plotted, the energy consumed between two time points is equivalent to the area below the power curve between the two time points. Therefore, energy is associated with a time window, for example, an application instance. Figure 10.8 presents *average* power consumption and energy consumption examples for one instance of some applications from the

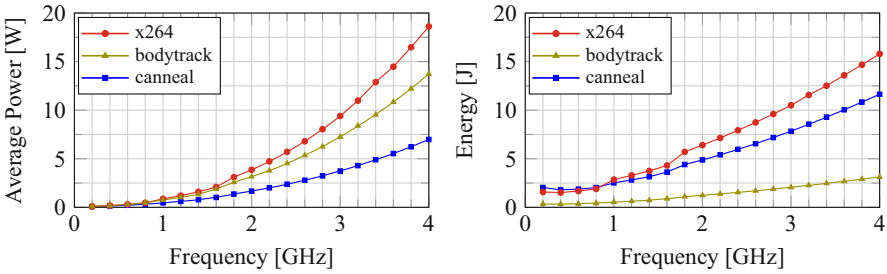


Fig. 10.8 Average power and energy values based on simulations in gem5 [4] and McPAT [31] for one instance of three applications from the PARSEC benchmark suite [3] executing a single thread on an *out-of-order* Alpha 21264 core

PARSEC benchmark suite [3] executing a single thread on an *out-of-order* Alpha 21264 core.

Previous work in the literature [27, 42] has shown that there exists a critical frequency for every application executing on a certain type of core which minimizes the energy consumption for execution. Namely, although executing at slow frequencies reduces the power consumption (due to the cubic relationship between dynamic power consumption and the frequency), it also prolongs the execution time of an application. Therefore, the critical frequency represents the frequency for which the energy savings achieved by reducing the power consumption (mainly savings in dynamic energy) are less significant than the corresponding increases in the energy consumption for prolonging the execution time (mainly due to leakage effects). In simple terms, this means that executing an application below its critical frequency for the corresponding type of core is not energy efficient, and it should hence be preferably avoided, even if it reduces the power consumption and meets the performance and timing constraints. The examples in Fig. 10.8 show the presence of such discussed critical frequency, where we can see that executing an application below 0.4 GHz (0.2 GHz in the figure) consumes more energy than executing it at 0.4 GHz.

Minimizing the overall energy consumption under timing (performance) constraints is a common optimization goal for real-time mobile systems, in which prolonging the battery lifetime is of major importance. Furthermore, on other battery-operated systems for which we can estimate the elapsed time between charging cycles (e.g., mobile phones), energy could also be used as a constraint, such that the system optimizes (maximizes) the overall performance under the battery's energy budget. Contrarily, it is very rare to optimize for power consumption, and thus power is mostly considered as a constraint, for example, to run the system under the given Thermal Design Power (TDP).

10.2.3 Temperature

Whenever some part of the chip is consuming power, it is also generating heat. Given that excessively high temperatures on the chip can cause permanent failures in transistors, maintaining the temperature throughout the chip below a certain threshold is of paramount importance. This issue is even more significant in modern chips due to voltage scaling limitations, which lead to increasing power densities across technology scaling generations and the so-called *dark silicon problem* [17, 37], that is, all parts of a chip cannot be simultaneously active at full speed. The use of a cooling solution (e.g., the combination of the thermal paste, the heat spreader, the heat sink, the cooling fan, etc.) and Dynamic Thermal Management (DTM) techniques is employed for such a purpose. DTM techniques are generally reactive (i.e., only become active after the critical temperature is reached or exceeded) and may power-down cores, gate their clocks, reduce their supply voltage and execution frequency, boost-up the fan speed, etc. Nevertheless, DTM techniques are generally aimed at avoiding the chip from overheating, not to optimize its performance. An abstract example of a DTM technique based on voltage and frequency scaling is presented in Fig. 10.9.

Although there exist some work which aims at reducing the peak temperature under performance constraints or at minimizing the thermal gradients in the chip, temperature is mostly considered as a constraint rather than a goal. Furthermore, thermal constraints tend to be the biggest limiting factor for performance optimization, especially in modern computing platforms in which power densities are ever increasing.

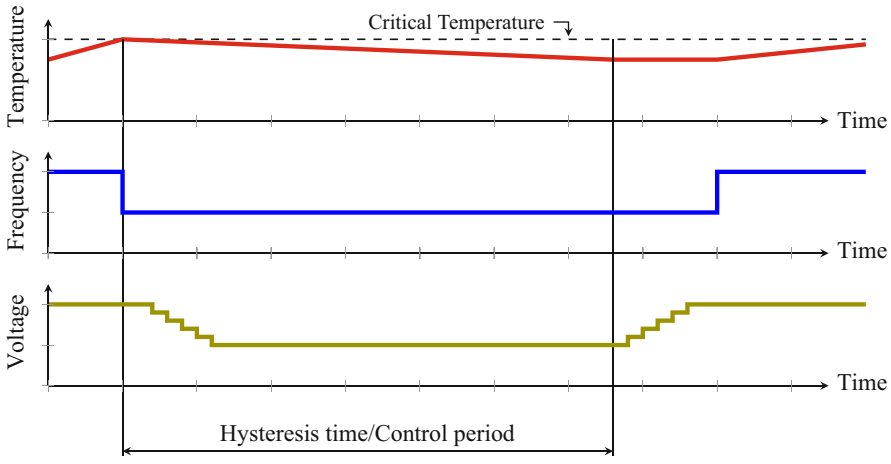


Fig. 10.9 Example of a DTM technique based on voltage and frequency scaling

The most widely adopted models used for thermal modeling in electronics are RC thermal networks, which are based on the well-known duality between thermal and electrical circuits [20]. In an RC thermal network, thermal conductances interconnect the thermal nodes among each other. Furthermore, there is a thermal capacitance associated with every thermal node that accounts for the transient effects in the temperatures, but there is no thermal capacitance associated with the ambient temperature as it is considered to be constant for long periods of time. The power consumption of cores and other elements corresponds to heat sources. In this way, the temperatures throughout the chip can be modeled as a function of the ambient temperature, the power consumptions inside the chip, and by considering the heat transfer among neighboring thermal nodes.

An example of a simplified RC thermal network for a chip with two cores is presented in Fig. 10.10. In the figure, T_1 and T_2 are voltages that represent the temperatures on core 1 and core 2, respectively. Voltages T_3 and T_4 represent the temperatures on the heat sink immediately above the cores. Current supplies p_1 and p_2 represent the power consumptions on each core. The thermal conductances b_{c}, $b_{\text{c-hs}}$, b_{hs}, and g_{amb} account for the heat transfer among the thermal nodes. Finally, the thermal capacitances of thermal node i are represented by capacitor a_i. By using Kirchoff's first law and linear algebra for the example in Fig. 10.10, we can derive a system of first-order differential equations as:

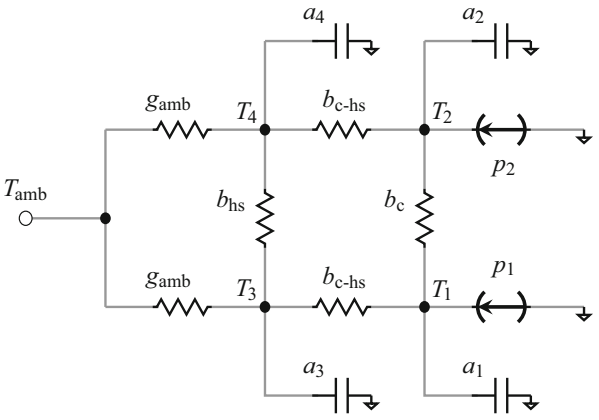


Fig. 10.10 Simple RC thermal network example for *two* cores (Figure from [46]), where we consider that cores are in direct contact with the heat sink and being the only connection between cores and the ambient temperature. A more detailed example would consider more layers between a core and the heat sink, for example, the ceramic packaging substrate, the thermal paste, and the heat spreader; and there would be more paths leading to the ambient temperature, for example, through the circuit board

$$\begin{cases} p_1 - (T_1 - T_3)\, b_{\text{c-hs}} + (T_2 - T_1)\, b_{\text{c}} - a_1 \dfrac{\mathrm{d}T_1}{\mathrm{d}t} = 0 \\ p_2 - (T_2 - T_4)\, b_{\text{c-hs}} - (T_2 - T_1)\, b_{\text{c}} - a_2 \dfrac{\mathrm{d}T_2}{\mathrm{d}t} = 0 \\ (T_1 - T_3)\, b_{\text{c-hs}} + (T_4 - T_3)\, b_{\text{hs}} - a_3 \dfrac{\mathrm{d}T_3}{\mathrm{d}t} - (T_3 - T_{\text{amb}})\, g_{\text{amb}} = 0 \\ (T_2 - T_4)\, b_{\text{c-hs}} - (T_4 - T_3)\, b_{\text{hs}} - a_4 \dfrac{\mathrm{d}T_4}{\mathrm{d}t} - (T_4 - T_{\text{amb}})\, g_{\text{amb}} = 0. \end{cases}$$

The system of first-order differential equations can be rewritten in matrix and vector form as:

$$\begin{bmatrix} b_{\text{c-hs}} + b_{\text{c}} & -b_{\text{c}} & -b_{\text{c-hs}} & 0 \\ -b_{\text{c}} & b_{\text{c-hs}} + b_{\text{c}} & 0 & -b_{\text{c-hs}} \\ -b_{\text{c-hs}} & 0 & b_{\text{c-hs}} + b_{\text{hs}} + g_{\text{amb}} & -b_{\text{hs}} \\ 0 & -b_{\text{c-hs}} & -b_{\text{hs}} & b_{\text{c-hs}} + b_{\text{hs}} + g_{\text{amb}} \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix} + \begin{bmatrix} a_1 & 0 & 0 & 0 \\ 0 & a_2 & 0 & 0 \\ 0 & 0 & a_3 & 0 \\ 0 & 0 & 0 & a_4 \end{bmatrix} \begin{bmatrix} T_1' \\ T_2' \\ T_3' \\ T_4' \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ 0 \\ 0 \end{bmatrix} + T_{\text{amb}} \begin{bmatrix} 0 \\ 0 \\ g_{\text{amb}} \\ g_{\text{amb}} \end{bmatrix}.$$

Therefore, RC thermal networks can serve as mathematical expressions to model the temperatures on the chip. In condensed matrix and vector form, the system of first-order differential equations of an RC thermal network is expressed as:

$$\mathbf{A}\mathbf{T}' + \mathbf{B}\mathbf{T} = \mathbf{P} + T_{\text{amb}}\mathbf{G},$$

where for a system with N thermal nodes, T_{amb} denotes the ambient temperature, matrix $\mathbf{A} = \left[a_{i,j}\right]_{N\times N}$ holds the values of the thermal capacitances (generally a diagonal matrix, since thermal capacitances are modeled to ground), matrix $\mathbf{B} = \left[b_{i,j}\right]_{N\times N}$ contains the values of the thermal conductances between vertical and lateral neighboring nodes (in $\left[\frac{\text{Watt}}{\text{Kelvin}}\right]$), column vector $\mathbf{T} = [T_i]_{N\times 1}$ represents the temperature on each node, column vector $\mathbf{T}' = \left[T_i'\right]_{N\times 1}$ represents the first-order derivative of the temperature on each node with respect to time, column vector $\mathbf{P} = [p_i]_{N\times 1}$ contains the values of the power consumption on every node, and column vector $\mathbf{G} = [g_i]_{N\times 1}$ contains the values of the thermal conductance between every node and the ambient temperature. In practice, the RC thermal network model of a chip and cooling solution can be modeled through profiling or by using a modeling tool like HotSpot [20].

10.2.4 Optimization Knobs

In order to achieve the abovementioned optimization goals under the corresponding constraints, efficient resource management techniques are required. Such techniques, however, are based on some basic hardware and software methods (commonly present in standard chip designs) which are used and exploited as optimization knobs. Among such optimization knobs, the most commonly used are thread-level selection, thread-to-core mapping and run-time task migration, Dynamic Power Management (DPM), and Dynamic Voltage and Frequency Scaling (DVFS). Specifically:

- **Thread-level selection** refers to selecting an appropriate level of parallelism for every application, which depending on the TLP of each application can have a major impact in the resulting performance, as seen in Sect. 10.2.1.
- **Thread-to-core mapping** involves to which specific core a thread is mapped to, both in terms of the type of core and the physical location of the core in the chip. In other words, the type of core to which a thread is mapped to is very important. Nevertheless, the selection of the physical location of the core is also a nontrivial issue, as this will have an impact on the performance (due to communication latencies among cores, potential link congestions, and the utilization of the shared resources) and also on the resulting temperature distribution (due to the heat transfer among cores, potentially creating or avoiding hotspots).
- **Run-time task migration** is simply the ability to migrate a task/thread from one core to another, at run-time. When migrating tasks at run-time, binary compatibility of tasks needs to be considered, given that, for example, different cores might have different ISAs, or a software task may be migrated to a reconfigurable fabric as a hardware implementation. Furthermore, it is also important to consider the non-negligible migration overheads, which could potentially result in larger performance penalties than benefits when applying too frequent migrations. For example, depending on the memory hierarchy, both instruction and data cache will experience many misses after a task is migrated from one core to another.
- **Dynamic Power Management (DPM)** refers to the dynamic power state selection of cores. For example, cores could be set to execution (active) mode, or they could be set to a low-power mode (e.g., clock gated, sleep, power gated, etc.). Every low-power mode has an associated power consumption and different latencies for entering and leaving each mode.
- **Dynamic Voltage and Frequency Scaling (DVFS)** refers to the ability to dynamically scale the voltage and/or frequency of a single core or a group of cores. Depending on the chip, voltage scaling and frequency scaling could be available at a per-core level, there could be only one global voltage and frequency, or it could be managed by groups of cores (i.e., clusters or voltage/frequency islands). For example, in Intel's Single Chip Cloud computer (SCC) [23], cores

are clustered into groups of eight cores that share the same voltage (dynamically set at run time), while the frequency of the cores can be selected every two cores, such that we can have up to four different frequencies inside each cluster of eight cores sharing a voltage.

10.3 Performance Optimization Under Power Constraints

As explained in Sect. 10.2, maximizing the overall system performance is generally the most commonly pursued optimization goal. In regard to the constraints, some resource management techniques consider power consumption, others consider temperature, and some consider both power and temperature. Furthermore, with respect to the power consumption, every system will have a physical power constraint which can, for example, be determined by the wire thickness or the supply voltage. Nevertheless, it is also very common to use power constraints as abstractions that allow system designers to indirectly deal with temperature. Namely, running the system under a given power constraint should presumably avoid violations of the thermal constraints. In line with such a concept, a power constraint aimed as a thermal abstraction is considered to be *safe* if satisfying it guarantees no thermal violations, and it is considered to be *efficient* if it results in temperatures that are not too far away from the critical temperature. Figure 10.11 shows an abstract example of a *safe* and *efficient* power budget, in which the maximum temperature throughout the chip remains just below the critical value when the system does not exceed the power budget.

The motivation for this approach is mainly to simplify the problem, given that proactive resource management techniques that directly deal with temperature are potentially more complex than those that only deal with power, mostly due to the heat transfer among cores and transient thermal effects.

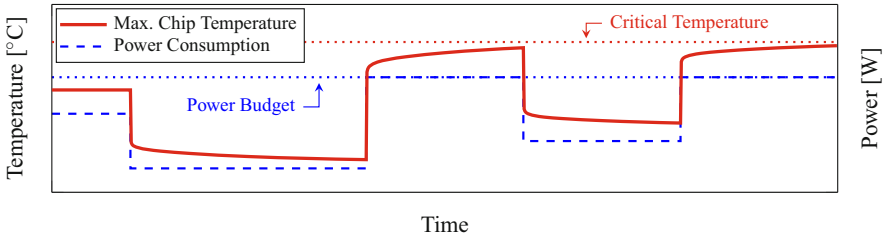


Fig. 10.11 Abstract example of a *safe* and *efficient* power budget

10.3.1 Traditional Per-Chip Power Constraints

The most common scheme in this direction (i.e., to have a power constraint as a thermal abstraction) is to use **TDP as a per-chip power constraint**, and there are several works in the literature aiming at performance optimization for such scenarios [30, 37, 52, 55].

Muthukaruppan et al. [37] propose a control-based framework that attempts to obtain the optimal trade-off between performance and power consumption, for homogeneous multi-core systems, while using TDP as a per-chip power constraint. DVFS and task migrations are used at different levels, specifically, at a task level, at a cluster level, and on the chip controllers. The controllers are coordinated such that they can throttle down the power consumption in case TDP is violated and to map tasks to cores in order to optimize the overall performance.

Also for overall performance optimization on homogeneous systems, Raghunathan et al. [52] try to exploit process variations between cores as a means for choosing the most suitable cores for every application. Their results show that, mostly due to the proportional increment of the process variations, the performance efficiency can potentially be increased along with the increase in the dark silicon area.

Sartori and Kumar [55] focus on maximizing many-core processor throughput for a given peak power constraint. It proposes three design-time techniques: mapping the power management problem to a knapsack problem, mapping it to a genetic search problem, and mapping it to a simple learning problem with confidence counters. These techniques prevent power from exceeding the given constraint and enable the placement of more cores on a die than what the power constraint would normally allow.

Kultursay et al. [30] build a 32-core TFET-CMOS heterogeneous multi-core processor and present a run-time scheme that improves the performance of applications running on these cores, while operating under a given power constraint. The run-time scheme combines heterogeneous thread-to-core mapping, dynamic work partitioning, and dynamic power partitioning.

> However, using a single and constant value as a power constraint, either at a per-chip or per-core level, for example, TDP, can easily result in thermal violations or significantly underutilized resources on multi-/many-core systems. This effect and a solution are discussed in Sect. 10.3.2.

10.3.2 Efficient Power Budgeting: Thermal Safe Power

For a system with 16 cores (*simple in-order* Alpha 21264 cores in 45 nm, simulated with McPAT [31]) and HotSpot's default cooling solution [20], Fig. 10.12 shows

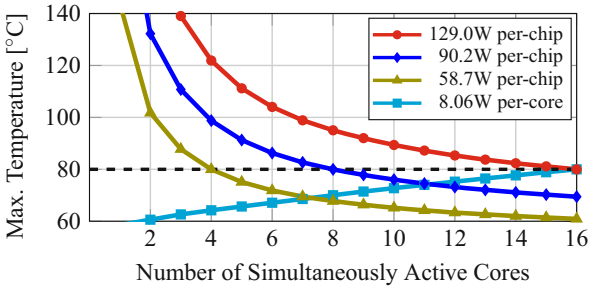


Fig. 10.12 Maximum steady-state temperature among all cores (DTM deactivated) as a function of the number of active cores, when using different *single* and *constant* per-chip and per-core power budgets

the maximum temperature among all cores (in the steady state, for concentrated mappings, and with DTM deactivated) as a function of the number of simultaneously active cores when using several traditional per-chip or per-core power budgets. Assuming that the critical temperature in this case is 80 °C, the figure shows that (for these specific concentrated mappings) there is only one point for each power budget in which the maximum temperature on the chip matches the critical value. For the other number of active cores, the temperature is either below or above the threshold, meaning that the power budget was not efficient or that it was not safe, respectively.

Therefore, the **Thermal Safe Power (TSP)** [47] power budget concept is introduced, proposing a *safe and efficient* alternative. The idea behind TSP is to have a per-core power constraint that depends on the number of active cores, rather than considering a single and constant value. Executing cores at power levels that satisfy TSP can result in a higher overall system performance when compared to traditional per-chip and per-core power budgeting solutions while maintaining the temperature throughout the chip below the critical value. Based on the RC thermal network of a given chip and its cooling solution, TSP can be computed at design time in order to obtain safe power constints for the worst-case mappings (namely, concentrated mappings promoting hotspots) as shown in the example in Fig. 10.13a, thus allowing the system to make thread-level selection decisions independent of the thread-to-core mapping. Furthermore, TSP can also be computed at run time for a (given) specific mapping of active cores, such that the system can further optimize the power budget for dispersed core mappings, for example, as shown in Fig. 10.13b.

Generally, as the number of active cores increases, the TSP power constraints decrease (as seen in Fig. 10.14), which in turn translates to executing cores at lower voltage and frequency levels. In this way, TSP derives a very simple relation between the number of active cores in a processor and their (application dependent) maximum allowed voltage and frequency levels for execution.

The major limitation with the techniques discussed in this section (both traditional power constraints and TSP) is that power is generally not easily measured in practical systems, mainly due to the lack of per-core power meters. In order to

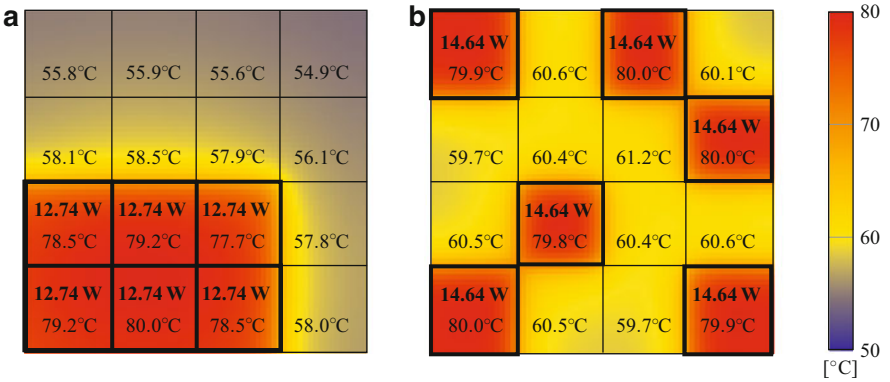


Fig. 10.13 Example of TSP for two different mappings for a maximum temperature of 80 °C (Figure from [46]). *Top numbers* are the power consumptions of each active core (boxed in *black*). *Bottom numbers* are the temperatures in the center of each core. Detailed temperatures are shown according to the *color bar*. (**a**) Concentrated mapping example with 6 active cores. (**b**) Distributed mapping example with 6 active cores

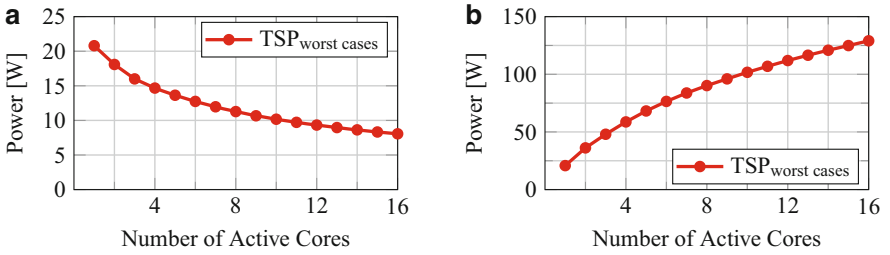


Fig. 10.14 Example of TSP for the worst-case core mappings. Per-chip values are estimated by multiplying per-core values with the number of active cores. (**a**) Per-core budget. (**b**) Estimated per-chip budget

address this issue, there are some works in the literature that attempt to estimate power consumption by measuring performance counters and core utilization [32]. Otherwise, extensive design-time application profiling with potential run-time refinement is required to estimate the power consumption of different applications running on different types of cores.

10.4 Performance Optimization Under Thermal Constraints

A different approach is to avoid using a power constraint as a thermal abstraction (as discussed in Sect. 10.3) and rather deal with temperature directly, either through thermal models or by using thermal sensors.

Using thermal models that rely on power estimations (as seen in Sect. 10.2.3) merely adds complexity to the problem (by considering the heat transfer among cores and the transient temperature effects), and it does so by maintaining the same issues with regard to how power consumption can be estimated in practice. Although this might look as a downside, the motivation for using these techniques instead of those presented in Sect. 10.3 is to avoid possible pessimistic or unsafe scenarios that can exist when using power constraints as thermal abstractions. Furthermore, these techniques can be proactive in nature, and we discuss them in more detail in Sect. 10.4.1.

Techniques that directly measure temperature by using thermal sensors can potentially be more accurate and easier to implement, as it is very common to find many thermal sensors in modern processors, to the point of possibly having one thermal sensor for every core in the chip. In this way, techniques that rely on thermal sensors can avoid the need for power consumption estimation tools. Nevertheless, the problem with these techniques is that it is very hard to do proactive thermal management without having thermal models. Therefore, they are generally reactive techniques which exploit the available thermal headroom, commonly also known as **boosting**, as discussed in Sect. 10.4.2.

10.4.1 Techniques Based on Thermal Modeling

Khdr et al. [28] propose a design-time dark silicon-aware resource management technique based on dynamic programming, called **DsRem**, which attempts to distribute the processors resources among different applications. Based on extensive application profiling, DsRem determines the number of active/dark cores and their voltage/frequency levels by considering the TLP and ILP characteristics of the applications, in order to maximize the overall system performance. Specifically, DsRem will attempt to map applications with high TLP by using a large number of cores executing at low voltage/frequency levels while mapping applications with high ILP to a small number of cores executing at high voltage/frequency levels. Applications that exhibit both high TLP and high ILP will potentially be mapped to a large number of cores executing at high voltage/frequency levels whenever possible.

Another work in the literature proposes a variability-aware dark silicon manager, called **DaSiM** [58]. In order to optimize the system performance under a thermal constraint, the idea behind DaSiM is to exploit the so-called dark silicon patterning in tandem with the core-to-core leakage power variations. Different dark silicon patterns denote different spatiotemporal decisions for the power state of the cores, namely, which cores to activate and which to put in low-power mode. These patterns directly influence the thermal profile on the chip due to improved heat dissipation, enabling to activate more cores to facilitate high-TLP applications and/or boosting certain cores to facilitate high-ILP applications. In order to enable run-time optimizations, DaSiM employs a lightweight run-time temperature prediction mechanism that estimates the chip's thermal profile for a given candidate solution.

Hanumaiah et al. [16] propose a thermal management technique based on RC thermal networks which attempts to minimize the latest completion time among the applications. This technique first derives an optimal solution by using convex optimization, which has a high time complexity and is therefore only suited for design-time decisions. Nevertheless, the structure of certain matrices in the convex optimization formulation can be exploited in order to have an approximate solution which is 20 times faster than the convex optimization approach. The implementation of such a technique for run-time adaptations is however debatable, as the experiments in [16] show that it may require more than 15 ms to compute the voltage and frequency levels of *each* core, which is generally not fast enough for run-time usage in multi-/many-core systems.

Pagani et al. [48] presents **seBoost**, a run-time boosting technique (see Sect. 10.4.2) based on analytical transient temperature estimation on RC thermal networks [45]. This technique attempts to meet run-time performance requirement surges, by executing the cores mapped with applications that require boosting at the specified voltages and frequencies while throttling down the cores mapped with application of lower priority. The performance losses of the low-priority applications are minimized by choosing the throttling down levels such that the critical temperature is reached precisely when the boosting interval is expected to expire. Furthermore, in order to select the throttle down levels of the cores mapped with the low-priority applications, seBoost performs a Design Space Exploration (DSE) but limiting the number of evaluated combinations by using a binary search like approach proportional to the nominal voltage/frequency operation levels on every core. A limitation of seBoost is that it assumes that the thread to core is given as an input, that is, it requires mapping decisions to be known a priori. Therefore, similar to the boosting techniques later explained in Sect. 10.4.2, seBoost needs to rely on another resource management technique to do the thread-level selection of applications and the mapping of threads to cores, such that it can then handle the boosting decisions and exploit the available thermal headroom.

10.4.2 Boosting Techniques

Boosting techniques have been widely adopted by chip manufacturers in commercial multi-/many-core systems, mostly because they provide the ability to exploit the existing thermal headroom in order to optimize the performance of a group of cores at run-time. Basically, by using DVFS, boosting techniques allow the system to execute some cores at high voltage and frequency levels during short intervals of time, even if this means exceeding standard operating power budgets (e.g., TDP), such that the system can optimize its performance under a thermal constraint. Given that executing at high voltage and frequency levels increases the power consumption in the chip, boosting techniques will incur in increments to the chip's temperature through time. Because of this increase in the temperature, once any part of the chip reaches a critical (predefined) temperature, the system should return to nominal operation (requiring some cool-down time before another

boosting interval is allowed) or use a closed loop control-based technique in order to oscillate around the critical temperature (allowing the boosting interval to be prolonged indefinitely) [48].

Boosting techniques generally do not aim at selecting the number of threads in which to parallelize applications, to make thread-to-core mapping decisions, or to migrate tasks between cores. Therefore, they are mostly well suited to exploit any available thermal headroom left by some other resource management technique (e.g., a thread-level selection and thread-to-core mapping technique based on a pessimistic per-chip power budget), in order to increase the system performance at run-time.

Intel's Turbo Boost [6, 8, 21, 22, 53] allows for a group of cores to execute at high voltage and frequency levels whenever there exists some headroom within power, current, and temperature constraints. In other words, when the temperature, power, and current are below some given values, the cores boost their voltage and frequency in single steps (within a control period) until it reaches a predetermined upper limit according to the number of active cores. Similarly, when either the temperature, power, or current exceeds the constraints, the cores reduce their voltage and frequency in single steps (also within a control period) until the corresponding constraints are satisfied or until the nominal voltage and frequency levels are reached. Although boosting to very high voltage and frequencies has an undesired effect on the power consumption (due to the cubic relationship between frequency and dynamic power consumption), Turbo Boost exploits the thermal capacitances of the thermal model knowing that, although there will be a temperature increase, this increase will require some time to reach the critical temperature rather than reach it immediately after the change in power. Figure 10.15 shows an example of Turbo

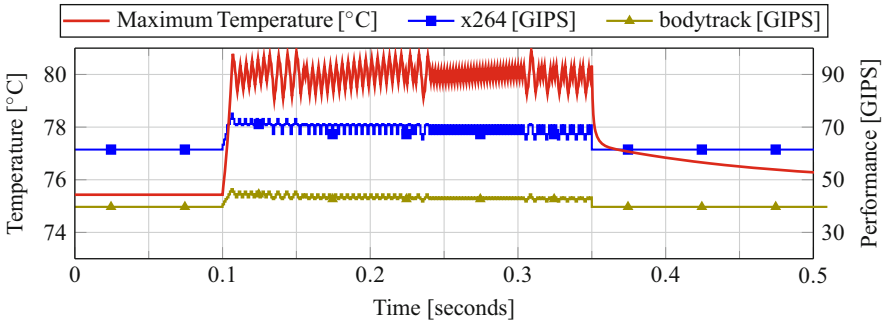


Fig. 10.15 Turbo Boost [6] example. The *red line* shows the maximum temperature among all cores (*left axis*). The performance of the applications is measured in Giga-Instruction Per Second (GIPS)

Boost's behavior based on simulations conducted in gem5 [4], McPAT [31], and HotSpot [20] for executing two applications from the PARSEC benchmark suite [3] in a multi-core processor with 16 *out-of-order* Alpha 21264 cores (each application running *eight* parallel dependent threads, one thread per core), under a temperature constraint of 80 °C.

Computational sprinting [51] is another boosting technique which proposes optimizing performance at run time via parallelism, by activating cores that are normally off (i.e., power gated) during short bursts of time (typically shorter than 1 s). Due to the cubic relationship between frequency and dynamic power consumption, computational sprinting intentionally discourages boosting through DVFS. Contrarily, it is motivated by the (ideally) linear relationship between performance and power expected when the system activates several cores at the same voltage and frequency levels. However, although this is a very valid point, the latency for waking up cores from low-power modes and the correspondent thread migrations can potentially result in significant overheads, especially when taking into consideration the short duration of the sprinting periods. Because of this, Turbo Boost will generally result in a higher overall system performance than computational sprinting.

10.5 Energy Optimization Under Performance Constraints

Energy-efficient scheduling and power management to minimize the overall energy consumption for **homogeneous** multi-core systems has been widely explored for real-time embedded systems with **per-core DVFS**, for example, [2, 9, 10, 36, 65]. Chen and Thiele [10] present a theoretical analysis of the Largest Task First (LTF) strategy, proving that, in terms of energy consumption, using LTF for task mapping results in solutions with approximation factors (particularly, the analytical worst-case ratio between the optimal solutions and the algorithms of interest) that depend on the hardware platforms. Moreover, [10, 65] propose polynomial-time algorithms that derive task mappings which attempt to execute cores at their critical frequency. For the special case in which there are uniform steps between the available core frequencies and also negligible leakage power consumption (a very restricting assumption), the work in [36] presents an algorithm that requires polynomial time for computing the optimal voltage and frequency assignments. Nevertheless, although having per-core DVFS can be very energy-efficient, Herbert and Marculescu [18] conducts extensive VLSI circuit simulations suggesting that it suffers from complicated design problems, making it costly for implementation. Therefore, assuming global DVFS, or DVFS at a cluster level (i.e., groups of cores or voltage/frequency islands), is much more realistic for practical systems, as seen in [23, 54].

For **homogeneous** systems with one global supply voltage and frequency, also referred to as **global DVFS**, Yang et al. [66] provide energy optimization solutions for systems with negligible leakage power consumption and frame-based real-time tasks (all tasks have the same arrival time and period). These are both very restricting

assumptions, and, therefore, the work in [12, 57] relaxes them in order to consider periodic real-time tasks (tasks have different arrival times and periodicity) with non-negligible leakage power consumption and non-negligible overhead for turning cores to low-power modes. Specifically, Seo et al. [57] proposes to dynamically balance the task loads of multiple cores and efficiently select the number of active cores, such that the power consumption for execution is minimized and the leakage power consumption for low workloads is reduced. Devadas and Aydin [12] first decide the number of active cores, and the voltage and frequencies of such cores are decided in a second phase. However, [12] does not provide theoretical analysis for the approximation factor of their proposed approach in terms of energy optimization. Furthermore, the basic ideas of [12] are used in [41] to theoretically derive the approximation factor, in terms of energy minimization, of the so-called Single Frequency Approximation (SFA) scheme. SFA is a simple strategy for selecting the DVFS levels on individual clusters. Particularly, after the tasks are assigned to clusters and cores, SFA uses a single voltage and frequency for all the cores in the cluster, specifically, the lowest DVFS level that satisfies the timing constraints of all tasks mapped to the cluster. Given that different tasks are assigned to different cores, not all cores in a cluster will have the same frequency requirements to meet the timing constraints. Therefore, the DVFS level of the cluster is defined by the core with the highest frequency demand in the cluster. Figure 10.16 presents an example of a cluster with four cores using SFA. The analysis of SFA is extended in [42] and [43] in order to consider the task partitioning phase.

For **homogeneous** systems with multiple **clusters of cores sharing their voltage and frequency**, there exist several heuristic algorithms [15, 29, 39, 44, 64], and

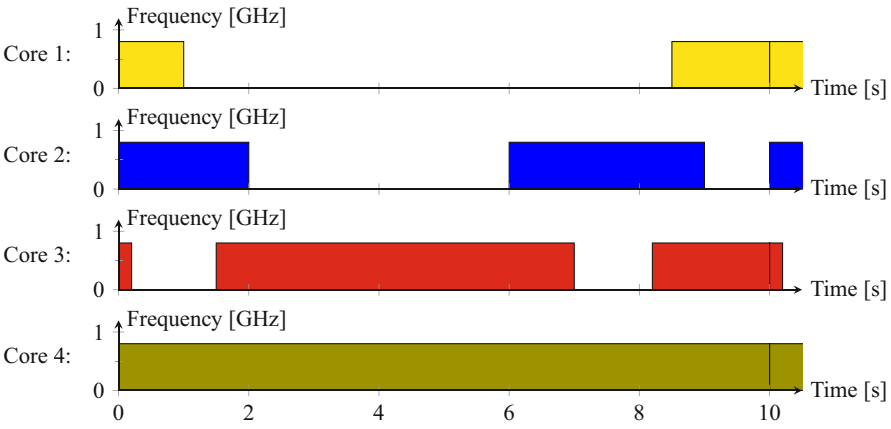


Fig. 10.16 SFA example for a cluster with four cores. The hyper-period of all tasks (i.e., the least common multiple among all periods of all tasks) is 10 s. To meet all deadlines, the frequency demand of the cores are 0.2, 0.4, 0.6, and 0.8 GHz. Hence, the single frequency is set to 0.8 GHz. In order to save energy, cores go individually to sleep when there is no workload on their ready queues

also an optimal dynamic programming solution [44], among which [13, 39, 44, 64] use SFA to choose the voltage and frequency of individual clusters. Particularly, Kong et al. [29] present a heuristic which first chooses the number of active clusters and then partitions the tasks by using the LTF task partitioning strategy. The task model used in [29] is later extended in [15] in order to consider shared resources and a synchronized version of LTF that considers that only one task can access specific resources at any given time instant. An Extremal Optimization meta-Heuristic (EOH) that considers a task graph and the communication costs among tasks is presented in [39], with the limitation that only one task can be assigned to each core. A heuristic based on genetic algorithms is presented in [64], where the energy consumption is gradually optimized in an iterative process through the selection, crossover, and mutation operations.

The work in [44] presents an optimal dynamic programming algorithm for given task sets (i.e., the tasks are already partitioned into cores, but the specific cores are not yet selected), called DYVIA, and suggests to use LTF for the task partitioning phase. Specifically, the authors of [44] first prove that when the average power consumption of different tasks executing at a certain DVFS level are equal or very similar and when the highest cycle utilization task sets in every cluster are given (i.e., when the DVFS levels of operation for every cluster are known), then the optimal solution will assign the highest cycle utilization task sets to the clusters running at the lowest possible DVFS levels, while still guaranteeing that the deadlines of all tasks are satisfied, as illustrated in Fig. 10.17. Furthermore, based on such a property, the DYVIA algorithm is able to reduce the number of combinations evaluated during its internal Design Space Exploration (DSE). For example, for a system with three clusters and three cores per cluster, if when finding the task sets to be assigned to cluster 3 we assume that the highest cycle utilization task set (i.e., $\mathbf{T}_{12}$) is always assigned to cluster 3, there are in total $\binom{8}{2} = 28$ possible combinations for selecting the other two task sets to be assigned to cluster 3. However, as shown in the example

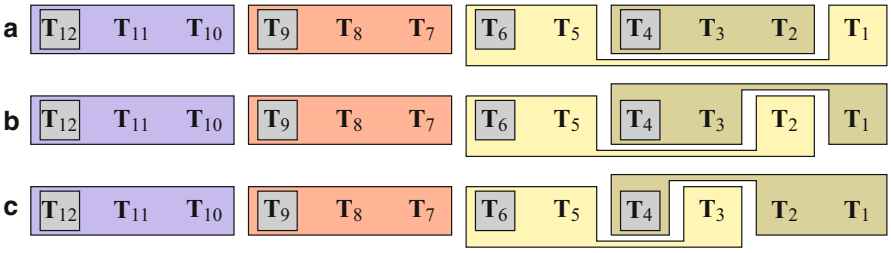


Fig. 10.17 Examples of possible task set assignments, for a chip with four clusters and three cores per cluster, where the task sets are increasingly ordered according to their cycle utilization. The figure shows the three possible assignments when the highest cycle utilization task sets in the clusters are $\mathbf{T}_4$, $\mathbf{T}_6$, $\mathbf{T}_9$, and $\mathbf{T}_{12}$ (boxed in *gray*), for which [44] proves that combination (**a**) minimizes the energy consumption

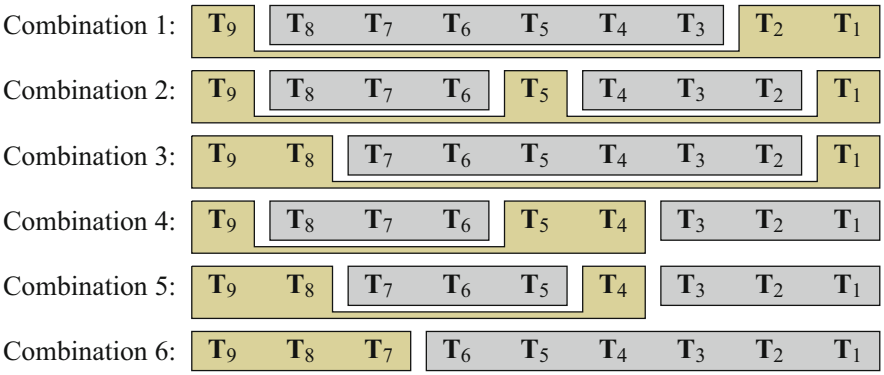


Fig. 10.18 Example of the potentially optimal combinations evaluated by the DYVIA algorithm [44], for a system with three clusters and three cores per cluster, where the task sets are increasingly ordered according to their cycle utilization. Each combination corresponds to the possible task sets to assign in cluster 3, for which DYVIA evaluates the resulting energy consumption, returning the minimum energy among all evaluated cases. In the figure, the task sets assigned to cluster 3 in a combination are boxed in *green*, and the resulting subproblems are colored in *gray*

in Fig. 10.18, for such case, DYVIA is able to reduce the design space, such that it only needs to evaluate *six potentially optimal combinations* in order to find the optimal assignment for cluster 3. DYVIA then finds the optimal assignment for cluster 2 by solving the associated subproblems.

There are several works focusing on **heterogeneous** systems with **per-core DVFS**. For example, Yang et al. [67] present a dynamic programming approach that uses trimming by rounding, in which the rounding factor can trade quality (in terms of energy optimization) of the derived solution with the total execution time of the algorithm.

There is also some work focusing on energy optimization for the more general model of **heterogeneous** multi-core systems with **clusters of cores sharing their voltage and frequency**, for example, [13, 38]. Muthukaruppan et al. [38] present a distributed framework based on price theory which attempts to minimize the overall power consumption, not the overall energy. Therefore, with the existence of critical frequencies, such a framework may fail to minimize the energy consumption when executing at very low frequencies (even when all performance constraints are satisfied). Moreover, this approach is not well suited for real-time systems, as it does not guaranty that all real-time tasks meet their deadlines, and only a *best effort* can be accomplished. Elewi et al. [13] propose a task partitioning and mapping scheme called Equally-Worst-Fit-Decreasing (EWFD), which attempts to balance the total utilization in every cluster. However, EWFD is an overly simplistic heuristic that assumes that executing different tasks on a given core type and frequency consumes equivalent power, which is not true in real systems as already observed in Fig. 10.8.

10.6 Hybrid Resource Management Techniques

Sharifi et al. [59] propose a joint temperature and energy management hybrid technique for heterogeneous Multi-Processor Systems-on-Chips (MPSoCs). In case that the system becomes underutilized, the technique focuses on energy minimization while satisfying the performance demands and thermal constraints. When the performance demands of the applications become higher, satisfying the thermal constraints has higher priority than minimizing energy, and thus the proposed technique applies a thermal balancing policy. This work includes a design-time application profiling phase that characterizes possible incoming tasks. It also derives different DVFS levels that balance the temperature throughout the chip for several performance demands. Then, at run-time, the technique integrates DVFS (including the design-time analysis) with a thread-to-core assignment strategy that is performance and temperature aware. When the performance demands can be satisfied only in some cores, the technique chooses which cores to power gate in order to minimize energy.

Ykman-Couvreur et al. [68] present a hybrid resource management technique for heterogeneous systems that aims at maximizing the overall Quality of Service (QoS) under changing platform conditions. In the design-time phase, by using an automated design-space exploration tool, the technique derives a set of Pareto-optimal application configurations under given QoS requirements and optimization goals. Then, the run-time resource management dynamically switches between the predefined configurations evaluated at design-time.

Singh et al. [60] present a hybrid management technique for mapping throughput-constrained applications on generic MPSoCs. The technique first performs design-time analysis of the different applications in order to derive multiple *resources/cores vs. throughput* trade-off points, therefore performing all the compute intensive analysis and leaving a minimum pending computation for the run-time phase. The run-time mapping strategy then selects the best point according to the desired throughput and available resources/cores.

Schor et al. [56] present a scenario-based technique for mapping a set of applications to a heterogeneous many-core system. The applications are specified as Kahn process networks. A finite state machine is used to coordinate the execution of the applications, where each state represents a scenario. During design-time analysis, the technique first precomputes a set of optimal mappings. Then, at run-time, hierarchically organized controllers monitor behavioral events and apply the precomputed mappings to start, stop, resume, and pause applications according to the finite state machine. In order to handle architectural failures, the technique allocates spare cores at design-time, such that the run-time controllers can move all applications assigned to a faulty physical core to a spare core. Given that this does not require additional design-time analysis, the proposed technique has a high responsiveness to failures.

Acknowledgments This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre *Invasive Computing* [SFB/TR 89] http://invasic.de.

References

1. Al Faruque MA, Krist R, Henkel J (2008) ADAM: run-time agent-based distributed application mapping for on-chipcommunication. In: Proceedings of the 45th IEEE/ACM design automation conference (DAC), pp 760–765. doi:10.1145/1391469.1391664
2. Aydin H, Yang Q (2003) Energy-aware partitioning for multiprocessor real-time systems. In: Proceedings of 17th international parallel and distributed processing symposium (IPDPS), pp 113–121
3. Bienia C, Kumar S, Singh JP, Li K (2008) The PARSEC benchmark suite: characterization and architectural implications. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques (PACT), pp 72–81
4. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The gem5 simulator. ACM SIGARCH Comput Archit News 39(2):1–7
5. Carlson TE, Heirman W, Eyerman S, Hur I, Eeckhout L (2014) An evaluation of high-level mechanistic core models. ACM Trans Archit Code Optim (TACO) 11(3):28:1–28:25. doi:10.1145/2629677
6. Casazza J (2009) First the tick, now the tock: intel microarchitecture (Nehalem). White paper, Intel Corporation
7. Ceng J, Castrillon J, Sheng W, Scharwächter H, Leupers R, Ascheid G, Meyr H, Isshiki T, Kunieda H (2008) MAPS: an integrated framework for MPSoC application parallelization. In: Proceedings of the 45th IEEE/ACM design automation conference (DAC), pp 754–759. doi:10.1145/1391469.1391663
8. Charles J, Jassi P, Ananth NS, Sadat A, Fedorova A (2009) Evaluation of the Intel core i7 turbo boost feature. In: IISWC, pp 188–197
9. Chen JJ, Hsu HR, Kuo TW (2006) Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In: Proceedings of the 12th IEEE real-time and embedded technology and applications symposium (RTAS), pp 408–417
10. Chen JJ, Thiele L (2010) Energy-efficient scheduling on homogeneous multiprocessor platforms. In: Proceedings of the ACM symposium on applied computing (SAC), pp 542–549
11. Choi J, Oh H, Kim S, Ha S (2012) Executing synchronous dataflow graphs on a SPM-based multicore architecture. In: Proceedings of the 49th IEEE/ACM design automation conference (DAC), pp 664–671. doi:10.1145/2228360.2228480
12. Devadas V, Aydin H (2010) Coordinated power management of periodic real-time tasks on chip multiprocessors. In: Proceedings of the international conference on green computing (GREENCOMP), pp 61–72
13. Elewi A, Shalan M, Awadalla M, Saad EM (2014) Energy-efficient task allocation techniques for asymmetric multiprocessor embedded systems. ACM Trans Embed Comput Syst (TECS) 13(2s):71:1–71:27
14. Grenat A, Pant S, Rachala R, Naffziger S (2014) 5.6 adaptive clocking system for improved power efficiency in a 28nm x86-64 microprocessor. In: IEEE international solid-state circuits conference digest of technical papers (ISSCC), pp 106–107
15. Han JJ, Wu X, Zhu D, Jin H, Yang L, Gaudiot JL (2012) Synchronization-aware energy management for vfi-based multicore real-time systems. IEEE Trans Comput (TC) 61(12):1682–1696

16. Hanumaiah V, Vrudhula S, Chatha KS (2011) Performance optimal online DVFS and task migration techniques for thermally constrained multi-core processors. Trans Comput Aided Des Integr Circuits Syst (TCAD) 30(11):1677–1690
17. Henkel J, Khdr H, Pagani S, Shafique M (2015) New trends in dark silicon. In: Proceedings of the 52nd ACM/EDAC/IEEE design automation conference (DAC), pp 119:1–119:6. doi:10.1145/2744769.2747938
18. Herbert S, Marculescu D (2007) Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In: Proceedings of the international symposium on low power electronics and design (ISLPED), pp 38–43
19. Howard J, Dighe S, Vangal S, Ruhl G, Borkar N, Jain S, Erraguntla V, Konow M, Riepen M, Gries M, Droege G, Lund-Larsen T, Steibl S, Borkar S, De V, Van Der Wijngaart R (2011) A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling. IEEE J Solid State Circuits 46(1):173–183. doi:10.1109/JSSC.2010.2079450
20. Huang W, Ghosh S, Velusamy S, Sankaranarayanan K, Skadron K, Stan MR (2006) HotSpot: a compact thermal modeling methodology for early-stage VLSI design. IEEE Trans VLSI Syst 14(5):501–513. doi:10.1109/TVLSI.2006.876103
21. Intel Corporation (2007) Dual-core intel xeon processor 5100 series datasheet, revision 003
22. Intel Corporation (2008) Intel turbo boost technology in Intel CoreTM microarchitecture (Nehalem) based processors. White paper
23. Intel Corporation (2010) SCC external architecture specification (EAS), revision 0.98
24. International technology roadmap for semiconductors (ITRS), 2011 edition. www.itrs.net
25. Jahn J, Pagani S, Kobbe S, Chen JJ, Henkel J (2013) Optimizations for configuring and mapping software pipelines in manycore. In: Proceedings of the 50th IEEE/ACM design automation conference (DAC), pp 130:1–130:8. doi:10.1145/2463209.2488894
26. Javaid H, Parameswaran S (2009) A design flow for application specific heterogeneous pipelined multiprocessor systems. In: Proceedings of the 46th IEEE/ACM design automation conference (DAC), pp 250–253. doi:10.1145/1629911.1629979
27. Jejurikar R, Pereira C, Gupta R (2004) Leakage aware dynamic voltage scaling for real-time embedded systems. In: Proceedings of the 41st design automation conference (DAC), pp 275–280
28. Khdr H, Pagani S, Shafique M, Henkel J (2015) Thermal constrained resource management for mixed ILP-TLP workloads in dark silicon chips. In: Proceedings of the 52nd ACM/EDAC/IEEE design automation conference (DAC), pp 179:1–179:6. doi:10.1145/2744769.2744916
29. Kong F, Yi W, Deng Q (2011) Energy-efficient scheduling of real-time tasks on cluster-based multicores. In: Proceedings of the 14th design, automation and test in Europe (DATE), pp 1–6
30. Kultursay E, Swaminathan K, Saripalli V, Narayanan V, Kandemir MT, Datta S (2012) Performance enhancement under power constraints using heterogeneous CMOS-TFET multicores. In: Proceedings of the 8th international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 245–254
31. Li S, Ahn JH, Strong R, Brockman J, Tullsen D, Jouppi N (2009) McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture (MICRO), pp 469–480
32. Li Y, Henkel J (1998) A framework for estimation and minimizing energy dissipation of embedded hw/sw systems. In: Proceedings of the 35th ACM/IEEE design automation conference (DAC), pp 188–193. doi:10.1145/277044.277097
33. Lin LY, Wang CY, Huang PJ, Chou CC, Jou JY (2005) Communication-driven task binding for multiprocessor with latency insensitive network-on-chip. In: The 15th Asia and South Pacific design automation conference (ASP-DAC), pp 39–44. doi:10.1145/1120725.1120739
34. Mallik A, Marwedel P, Soudris D, Stuijk S (2010) MNEMEE: a framework for memory management and optimization of static and dynamic data in MPSoCs. In: Proceedings of the international conference on compilers, architectures and synthesis for embedded systems (CASES), pp 257–258. doi:10.1145/1878921.1878959

35. Martin G (2006) Overview of the MPSoC design challenge. In: Proceedings of the 43rd IEEE/ACM design automation conference (DAC), pp 274–279. doi:10.1109/DAC.2006.229245
36. Moreno G, de Niz D (2012) An optimal real-time voltage and frequency scaling for uniform multiprocessors. In: Proceedings of the 18th IEEE international conference on embedded and real-time computing systems and applications (RTCSA), pp 21–30
37. Muthukaruppan T, Pricopi M, Venkataramani V, Mitra T, Vishin S (2013) Hierarchical power management for asymmetric multi-core in dark silicon era. In: DAC, pp 174:1–174:9
38. Muthukaruppan TS, Pathania A, Mitra T (2014) Price theory based power management for heterogeneous multi-cores. In: Proceedings of the 19th international conference on architectural support for programming languages and operating systems (ASPLOS), pp 161–176
39. Nikitin N, Cortadella J (2012) Static task mapping for tiled chip multiprocessors with multiple voltage islands. In: Proceedings of the 25th international conference on architecture of computing systems (ARCS), pp 50–62
40. Orsila H, Kangas T, Salminen E, Hämäläinen TD, Hännikäinen M (2007) Automated memory-aware application distribution for multi-processor system-on-chips. J Syst Archit 53(11):795–815. doi:10.1016/j.sysarc.2007.01.013
41. Pagani S, Chen JJ (2013) Energy efficiency analysis for the single frequency approximation (SFA) scheme. In: Proceedings of the 19th IEEE international conference on embedded and real-time computing systems and applications (RTCSA), pp 82–91. doi:10.1109/RTCSA.2013.6732206
42. Pagani S, Chen JJ (2013) Energy efficient task partitioning based on the single frequency approximation scheme. In: Proceedings of the 34th IEEE real-time systems symposium (RTSS), pp 308–318. doi:10.1109/RTSS.2013.38
43. Pagani S, Chen JJ, Henkel J (2015) Energy and peak power efficiency analysis for the single voltage approximation (SVA) scheme. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 34(9):1415–1428. doi:10.1109/TCAD.2015.2406862
44. Pagani S, Chen JJ, Li M (2015) Energy efficiency on multi-core architectures with multiple voltage islands. IEEE Trans Parallel Distrib Syst (TPDS) 26(6):1608–1621. doi:10.1109/TPDS.2014.2323260
45. Pagani S, Chen JJ, Shafique M, Henkel J (2015) MatEx: efficient transient and peak temperature computation for compact thermal models. In: Proceedings of the 18th design, automation and test in Europe (DATE), pp 1515–1520
46. Pagani S, Khdr H, Chen JJ, Shafique M, Li M, Henkel J (2016) Thermal safe power: efficient thermal-aware power budgeting for manycore systems in dark silicon. In: The dark side of silicon. Springer
47. Pagani S, Khdr H, Munawar W, Chen JJ, Shafique M, Li M, Henkel J (2014) TSP: thermal safe power – efficient power budgeting for many-core systems in dark silicon. In: The international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 10:1–10:10. doi:10.1145/2656075.2656103
48. Pagani S, Shafique M, Khdr H, Chen JJ, Henkel J (2015) seBoost: selective boosting for heterogeneous manycores. In: Proceedings of the 10th IEEE/ACM international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 104–113
49. Pinckney N, Sewell K, Dreslinski RG, Fick D, Mudge T, Sylvester D, Blaauw D (2012) Assessing the performance limits of parallelized near-threshold computing. In: 49th design automation conference (DAC), pp 1147–1152
50. Quan W, Pimentel AD (2015) A hybrid task mapping algorithm for heterogeneous MPSoCs. ACM Trans Embed Comput Syst (TECS) 14(1):14:1–14:25. doi:10.1145/2680542
51. Raghavan A, Luo Y, Chandawalla A, Papaefthymiou M, Pipe KP, Wenisch TF, Martin MMK (2012) Computational sprinting. In: Proceedings of the IEEE 18th international symposium on high-performance computer architecture (HPCA), pp 1–12
52. Raghunathan B, Turakhia Y, Garg S, Marculescu D (2013) Cherry-picking: exploiting process variations in dark-silicon homogeneous chip multi-processors. In: DATE, pp 39–44
53. Rotem E, Naveh A, Rajwan D, Ananthakrishnan A, Weissmann E (2012) Power-management architecture of the Intel microarchitecture code-named sandy bridge. IEEE Micro 32(2):20–27

54. Samsung Electronics Co., Ltd.: Exynos 5 Octa (5422). www.samsung.com/exynos
55. Sartori J, Kumar R (2009) Three scalable approaches to improving many-core throughput for a given peak power budget. In: International conference on high performance computing (HiPC), pp 89–98
56. Schor L, Bacivarov I, Rai D, Yang H, Kang SH, Thiele L (2012) Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Proceedings of the 15th international conference on compilers, architectures and synthesis for embedded systems (CASES), pp 71–80. doi:10.1145/2380403.2380422
57. Seo E, Jeong J, Park SY, Lee J (2008) Energy efficient scheduling of real-time tasks on multicore processors. IEEE Trans Parallel Distrib Syst (TPDS) 19(11):1540–1552. doi:10.1109/TPDS.2008.104
58. Shafique M, Gnad D, Garg S, Henkel J (2015) Variability-aware dark silicon management in on-chip many-core systems. In: Proceedings of the 18th design, automation and test in Europe (DATE), pp 387–392
59. Sharifi S, Coskun AK, Rosing TS (2010) Hybrid dynamic energy and thermal management in heterogeneous embedded multiprocessor SoCs. In: Proceedings of the Asia and South Pacific design automation conference (ASP-DAC), pp 873–878
60. Singh AK, Kumar A, Srikanthan T (2011) A hybrid strategy for mapping multiple throughput-constrained applications on MPSoCs. In: Proceedings of the 14th international conference on compilers, architectures and synthesis for embedded systems (CASES), pp 175–184. doi:10.1145/2038698.2038726
61. Smit L, Smit G, Hurink J, Broersma H, Paulusma D, Wolkotte P (2004) Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture. In: Proceedings of the IEEE international conference on field-programmable technology (FPT), pp 421–424. doi:10.1109/FPT.2004.1393315
62. Tan C, Muthukaruppan T, Mitra T, Ju L (2015) Approximation-aware scheduling on heterogeneous multi-core architectures. In: The 20th Asia and South Pacific design automation conference (ASP-DAC), pp 618–623
63. Weichslgartner A, Gangadharan D, Wildermann S, GlaßM, Teich J (2014) DAARM: design-time application analysis and run-time mapping for predictable execution in many-core systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 34:1–34:10. doi:10.1145/2656075.2656083
64. Wu X, Zeng Y, Han JJ (2013) Energy-efficient task allocation for VFI-based real-time multi-core systems. In: Proceedings of the international conference on information science and cloud computing companion (ISCC-C), pp 123–128
65. Xu R, Zhu D, Rusu C, Melhem R, Mossé D (2005) Energy-efficient policies for embedded clusters. In: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems (LCTES), pp 1–10
66. Yang CY, Chen JJ, Kuo TW (2005) An approximation algorithm for energy-efficient scheduling on a chip multiprocessor. In: Proceedings of the 8th design, automation and test in Europe (DATE), pp 468–473
67. Yang CY, Chen JJ, Kuo TW, Thiele L (2009) An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In: Proceedings of the 12th design, automation and test in Europe (DATE), pp 694–699
68. Ykman-Couvreur C, Hartmann PA, Palermo G, Colas-Bigey F, San L (2012) Run-time resource management based on design space exploration. In: Proceedings of the 8th IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 557–566. doi:10.1145/2380445.2380530

Part IV
Processor, Memory, and Communication Architecture Design

Reconfigurable Architectures

11

Mansureh Shahraki Moghaddam, Jae-Min Cho, and Kiyoung Choi

Abstract

Reconfigurable architecture is a computer architecture combining some of the flexibility of software with the high performance of hardware. It has configurable fabric that performs a specific data-dominated task, such as image processing or pattern matching, quickly as a dedicated piece of hardware. Once the task has been executed, the hardware can be adjusted to do some other task. This allows the reconfigurable architecture to provide the flexibility of software with the speed of hardware. This chapter discusses two major streams of reconfigurable architecture: Field-Programmable Gate Array (FPGA) and Coarse Grained Reconfigurable Architecture (CGRA). It gives a brief explanation of the merits and usage of reconfigurable architecture and explains basic FPGA and CGRA architectures. It also explains techniques for mapping applications onto FPGAs and CGRAs.

Acronyms

ALAP	As Late As Possible
ALM	Adaptive Logic Module
ALU	Arithmetic-Logic Unit
ASAP	As Soon as Possible
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
ASMBL	Advanced Silicon Modular Block
CCE	Configuration Cache Element
CDFG	Control-/Data-Flow Graph
CGRA	Coarse Grained Reconfigurable Architecture

M.S. Mansureh (✉) • J.-M. Cho • K. Choi
Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea
e-mail: mansureh@dal.snu.ac.kr; jaemincho@dal.snu.ac.kr; kchoi@dal.snu.ac.kr

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_12

CLB	Configurable Logic Block
DFG	Data-Flow Graph
DPR	Dynamic Partial Reconfiguration
DRAA	Dynamically Reconfigurable ALU Array
DRESC	Dynamically Reconfigurable Embedded System Compiler
DSP	Digital Signal Processor
EGRA	Expression Grained Reconfigurable Array
EMS	Edge Centric Modulo Scheduling
ESL	Electronic System Level
FDS	Force-Directed Scheduling
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GOPS	Giga Operations Per Second
GPP	General-Purpose Processor
HLS	High-Level Synthesis
II	Initiation Interval
ILP	Integer Linear Program
IMS	Iterative Modulo Scheduling
IOE	I/O Element
I/O	Input/Output
LAB	Logic Array Block
LE	Logic Element
LLVM	Low-Level Virtual Machine
LS	List Scheduling
LUT	Look-Up Table
MRRG	Modulo Resource Routing Graph
NoC	Network-on-Chip
NRE	Non-Recurring Engineering
PE	Processing Element
PLL	Phase Locked Loop
QEA	Quantum-inspired Evolutionary Algorithm
RCM	Reconfigurable Computing Module
RF	Register File
RTL	Register Transfer Level
SDF	Synchronous Data Flow
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SPKM	Split & Push Kernel Mapping
SPMD	Single Program, Multiple Data
SPM	Scratchpad Memory
STMD	Single Thread, Multiple Data
VLIW	Very Long Instruction Word
VLSI	Very-Large-Scale Integration

Contents

11.1 Why Reconfigurable Architectures?

General-Purpose Processors (GPPs) are programmable but not good in terms of performance (or execution time) when compared to Application-Specific Integrated Circuits (ASICs). ASICs are specialized circuits providing large amount of parallelism and thus allowing high performance implementation, but only for a specific application. An ASIC can contain just the right mix of functional units for a particular application and thus can be made fast and compact. They can make a very dense chip, which typically translates to high scalability. As technology has improved over the years, the maximum complexity (and hence functionality) possible in an ASIC has grown from several thousand gates to over millions of gates. But ASICS are not an economic choice for many embedded applications due to higher Non-Recurring Engineering (NRE) cost and longer time to market, except for very large volume applications. Reconfigurable computing systems like FPGAs and CGRAs as an intermediate architecture can provide both performance and flexibility. The performance is from the parallelism of the architecture, and the flexibility is from the configurability of the architecture. While FPGAs provide fine-grained (gate level) reconfigurability, CGRAs provide coarse-grained (register transfer level) reconfigurability.

The world of multimedia processing and telecommunication stack is characterized by increasing speed and performance needs. The required raw compute power has been fed by the ever increasing transistor densities enabled by innovations in the Very-Large-Scale Integration (VLSI) domain. However, this growth has to take into account increasing process/voltage/temperature variations, shorter time to market, and higher NRE cost. That is, it is required to achieve both higher

performance and more efficient design/manufacturing at the same time. These two conflicting requirements have made reconfigurable architectures a popular alternative implementation platform.

FPGA is the first successful reconfigurable architecture. The most popular SRAM-based FPGAs contain many programmable logic blocks that can be reprogrammed many times after manufacturing, although some FPGAs such as the one using the antifuse technology can be programmed only once. It can be used as a test bed to prototype a design before going for a final ASIC design. In this way, FPGAs can be reprogrammed as needed until the design is finalized. The ASIC can then be manufactured based on the FPGA design.

Since an FPGA is basically an array of gates, it also provides a large amount of parallelism and thus allows high performance implementation. Actually, the design phases of FPGAs and ASICs are quite similar except that ASICs lack post-silicon flexibility. ASICs require new fabrication for a new application, and thus fail to satisfy the market's critical time-to-market needs, and are, by definition, unable to satisfy the need for greater flexibility [96]. FPGAs are more flexible with the ability to rapidly implement or reprogram the logic. The general flexibility of an FPGA results in time-to-market advantages since it allows fast implementation of new functions as well as easy bug fixes. One thing to note, however, is that an ASIC is designed to be fully optimized to a specific application or a function. Compared to ASICs, FPGAs consume more power, take more area, and provide lower performance but have much lower NRE cost. Thus FPGAs are in general much more cost-effective than ASICs for low production volumes.

The increase of logic in an FPGA has enabled larger and more complex algorithms to be programmed into the FPGA. Furthermore, algorithms can be parallelized and implemented on multiple FPGAs resulting in highly parallel computing. The attachment of such an FPGA to a modern CPU over a high speed bus, like PCI express, has enabled the configurable logic to act more like an accelerator rather than a peripheral. This has brought reconfigurable computing into the high-performance computing sphere. Of course, the use of FPGAs requires creating the hardware design, which is a costly and labor-intensive task, although the vendors typically provide IP cores for common processing functions [13].

The reconfiguration granularity of CGRAs is larger than that of FPGAs. CGRAs typically have an array of simple Processing Elements (PEs), where the PEs are connected with each other through programmable interconnects. The functionality of each PE is also programmable. Compared to FPGAs, CGRAs have significant reduction in area and energy consumption due to much less amount of configuration memory, switches, and interconnects for programming. Furthermore, because of the low overhead of reconfiguration, CGRAs offer dynamic reconfiguration capabilities, which is not easy for FPGAs. That makes CGRAs attractive for area-constrained designs.

Processors are considered to be most flexible in that any kind of application with complicated control and data dependencies can be easily compiled and mapped onto the architecture. However, realizing the flexibility requires a rich set of instructions and the supporting hardware, which incurs a significant overhead in terms of area cost and power consumption. Moreover, with a single GPP, it is difficult to exploit

the parallelism in the application because of the complexity in the architecture. There have been abundant researches and developments to enhance the performance of GPPs by exploiting parallelism; actually, there have been startling progresses in architectures supporting instruction-level parallelism such as superscalar and Very Long Instruction Word (VLIW) architectures. However, it seems no longer possible to make such a progress in that direction due to the rapid growth of area cost and power consumption (area cost or number of transistors is less a concern today, but it still matters in many applications that consider cost, form factor, leakage current, etc.). GPPs, Digital Signal Processors (DSPs), and Application-Specific Instruction-set Processors (ASIPs) (For the details of ASIP, refer to ▸ Chap. 12, "Application-Specific Processors".) belong to this category, although DSPs and ASIPs are in general less flexible than GPPs.

On the other hand, a new architecture has come to importance, named as multi-core or many-core architecture depending on the number of processor cores integrated on a chip. The processor cores are connected by a bus, or by a Network-on-Chip (NoC) when there are too many cores to be connected by a bus. Such an architecture can exploit task-level parallelism through proper scheduling of tasks, while exploiting instruction-level parallelism available in a task if the processor cores are capable of doing it. Although such an architecture can better exploit parallelism with a better scalability, the processor cores are still very expensive, and the on-chip communications incur additional costs in terms of area and power consumption.

Different applications place unique and distinct demands on computing resources, and applications that work well on one processor architecture will not necessarily map well to another; this is true even for different phases of a single application. As yet another architecture, GPUs are inexpensive, commodity parallel devices with huge market penetration. They have already been employed as powerful accelerators for a large number of applications including games and 3D physics simulation. The main advantages of a GPU as an accelerator stem from its high memory bandwidth and a large number of programmable cores with thousands of hardware thread contexts executing programs in a Single Program, Multiple Data (SPMD) (The model of GPUs executing the same kernel code on multiple data is called differently in the literature. Examples other than SPMD include Single Instruction, Multiple Data (SIMD), Single Instruction, Multiple Threads (SIMT), and Single Thread, Multiple Data (STMD).) fashion. GPUs are flexible and relatively easy to program using high-level languages and APIs which abstract away hardware details. Changing functions in GPUs can be done simply via rewriting and recompiling code. However, this flexibility comes at a cost. For the flexibility, GPUs rely on the traditional von Neumann architecture that fetches instructions from memory, although the SPMD model can execute many threads in parallel to process many different data with a single-thread program fetch. Thus, when the application cannot generate many threads having the same program sequence, the architecture may result in waste of resources and inefficiency in terms of area cost and power consumption. Figure 11.1 briefly expresses the positioning of reconfigurable architectures in terms of efficiency versus flexibility compared to other technologies/architectures including ASIC, GPP, and others.

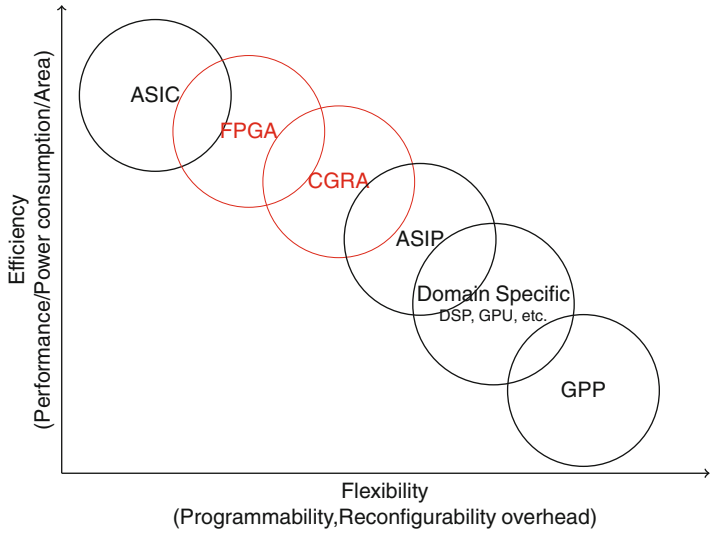


Fig. 11.1 Comparison between implementation platforms [20]

11.2 FPGA Architecture

Field-Programmable Gate Arrays (FPGAs) went far beyond the peripheral position in early days and are now occupying central positions in highly complex systems. Over the 30 years, FPGAs have increased capacity by more than a factor of 10,000 and increased speed by a factor of 100. Cost and energy consumption per unit function have also decreased by more than a factor of 1,000 [105]. An FPGA device provides millions of logic cells, megabytes of block memory, thousands of DSP blocks, and gigahertz of clock speed [72]. FPGAs are getting more complex with the advances in semiconductor technology and are now found in various systems, such as network, television, automobiles, etc., due to their merits compared to other state-of-the-art architectures. For example, an Altera Arria 10 FPGA [2] with DSP blocks that support both fixed and floating-point arithmetic can perform up to 1 TFLOPS [57]. An FPGA platform with four Virtex-5 FPGAs [114] offers performance comparable to a CPU or a GPU with 2.7–293 times better energy efficiency on the BLAS benchmark [46].

Also, since FPGAs have the capability of reconfiguration, multiple applications can be implemented on a small device, and thus the gap between FPGAs and ASIC designs in terms of area and power can be reduced [28]. FPGA has also established a large area of research on self-adaptive hardware systems. Self-adaptive hardware systems are capable of exchanging, updating, or extending hardware functionality during run time.

11.2.1 Building Blocks

FPGAs contain a large amount of logic elements and interconnects among them. There are also interconnects for clock distribution. The bitstream of configuration data is downloaded into an FPGA to configure the logic elements and the interconnects to implement a required behavior on the FPGA.

11.2.1.1 Logic Elements

The types of Logic Elements (LEs) are different depending on the manufacturing companies. Most common LE types are registers, Look-Up Table (LUT), block RAMs, and DSP units. Modern FPGAs combine such logic elements into a customized building block for architectural scalability. For example, the LUTs in Xilinx 7 series FPGA [113] can be configured as either a 6-input LUT with one output or two 5-input LUTs with separate outputs but common addresses or logic inputs. Each 5-input LUT output can optionally be registered in a flip-flop. Figure 11.2 shows a simplified diagram of a sub-block (dotted box) consisting of an LUT, two flip-flops, and several multiplexers. The logic circuit from Cin to Cout in the diagram is used to build a carry chain for an efficient implementation of an adder/subtracter. Four copies of such a sub-block form a slice (dashed box); thus a slice contains four LUTs and eight flip-flops in total together with multiplexers and carry logic. Among the eight flip-flops in a slice, four (one per LUT) can optionally be configured as latches. Two slices form a Configurable Logic Block (CLB); each slice in a CLB is connected to a switch matrix as shown in Fig. 11.3. One of the reasons for this specific CLB structure, which is common to Spartan-6 and Virtex-6, is to simplify design migration from the Spartan-6 and Virtex-6 families to the 7 series devices [115].

As shown in Fig. 11.4, the CLBs are arranged in columns in the 7 series FPGAs (see Fig. 11.3) to form the Advanced Silicon Modular Block (ASMBL) architecture, which uses flip-chip packaging to place pins anywhere (not only along the periphery). With the architecture, the number of I/O pins can be increased arbitrarily without increasing the array size as shown in Fig. 11.5 [112]. It also enhances on-chip power and ground distribution by allowing power and ground lines to be placed at proper locations in the chip as shown in Fig. 11.6. The ASMBL architecture enables an FPGA platform to optimize the mixture of resource columns to an application domain. In Fig. 11.4, for example, applications in domain A require lots of logic, some memory blocks, and small number of DSP blocks, and thus they fit well with platform A since it has a mixture of columns optimized to such applications. Each CLB block can be configured as a look-up table, distributed RAM, or a shift register.

Altera Stratix V FPGA [4] devices use a building block called enhanced Adaptive Logic Module (ALM) to implement logic functions more efficiently. The enhanced ALM has a fracturable LUT with eight inputs, two dedicated embedded adders, and four dedicated registers as shown in Fig. 11.7. The ALM in Stratix V packs 6% more

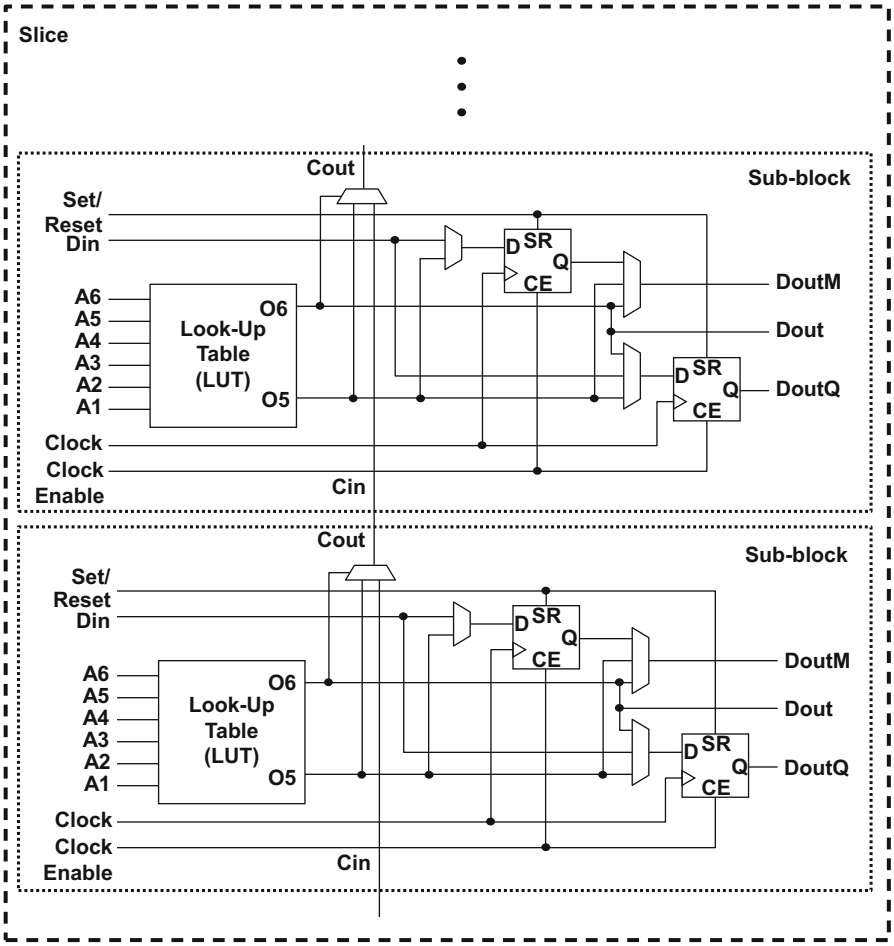


Fig. 11.2 Sub-blocks in a slice

logic compared to the previous-generation ALM found in Stratix IV devices. An ALM can implement some 7-input LUT-based function, any 6-input logic function, two independent functions with a smaller-sized LUT (such as two independent 4-input LUT-based functions), and two independent functions that share some inputs as shown in Fig. 11.8; this is the reason for calling LUT as a fracturable LUT. This enables Stratix V devices to maximize core performance at higher core logic utilization and provide easier timing closure for register-rich and heavily pipelined designs.

11.2.1.2 Interconnects

In Xilinx UltraScale architecture [72], extra connectivity (bypass connections shown in Fig. 11.9) eliminates the need to route through an LUT to gain access to the

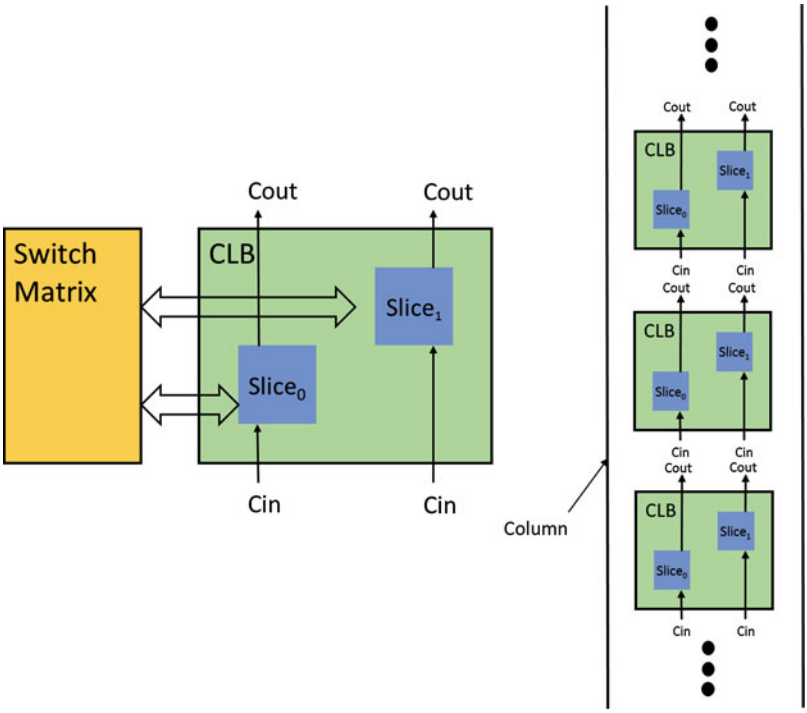


Fig. 11.3 CLB block diagram

associated flip-flops. The flip-flops in the CLB of the architecture benefit from several flexibility enhancements such as inversion attributes. The inversion attributes are used to change the active polarity of each pin. When set to 1, it changes the pin to behave active-low rather than active-high. Having more control signals with increased flexibility provides the software with additional flexibility to use all the resources within each CLB in the architecture.

Old FPGA generations have used central clock spine to distribute the various clocks throughout the FPGA. As a result, clock skew always grows larger when clock sources are away from the center of the device. In Xilinx UltraScale architecture, segmented clock networks allow the center of clock network of a logic block to be placed at the geometric center of the logic block. This technique reduces the clock skew and also improves the performance. The clock segments can also switch on and off when needed. This scheme eliminates unnecessary transistor switchings and reduces the amount of power required to run the on-chip clock networks.

The high-performance Altera Stratix architecture also consists of vertically arranged LEs, memory blocks, DSP blocks, and Phase Locked Loops (PLLs) that are surrounded by I/O Elements (IOEs) as depicted in Fig. 11.10. Speed-optimized interconnects and low-skew clock networks provide connectivity between these

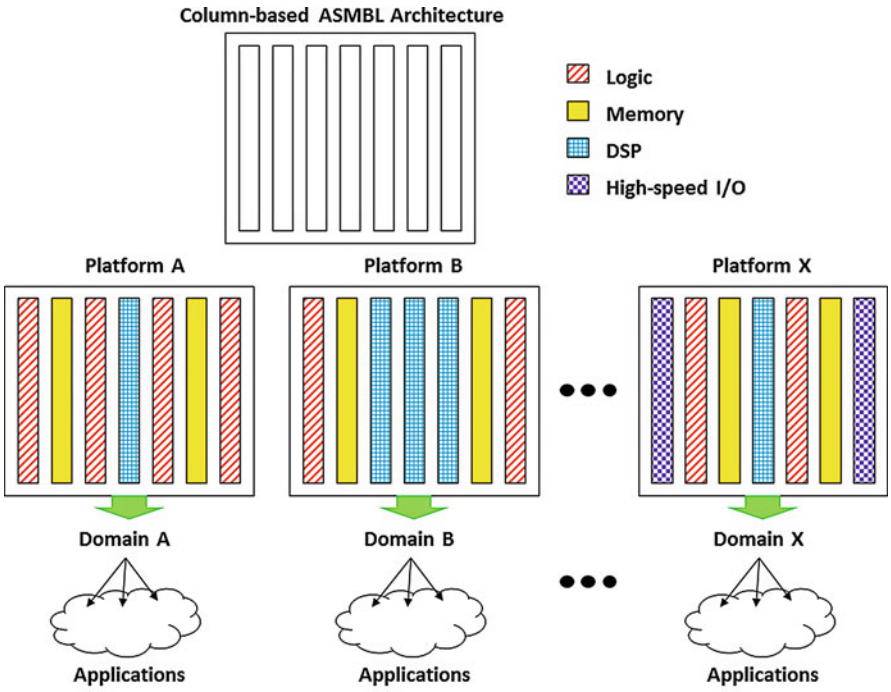


Fig. 11.4 Xilinx ASMBL architecture

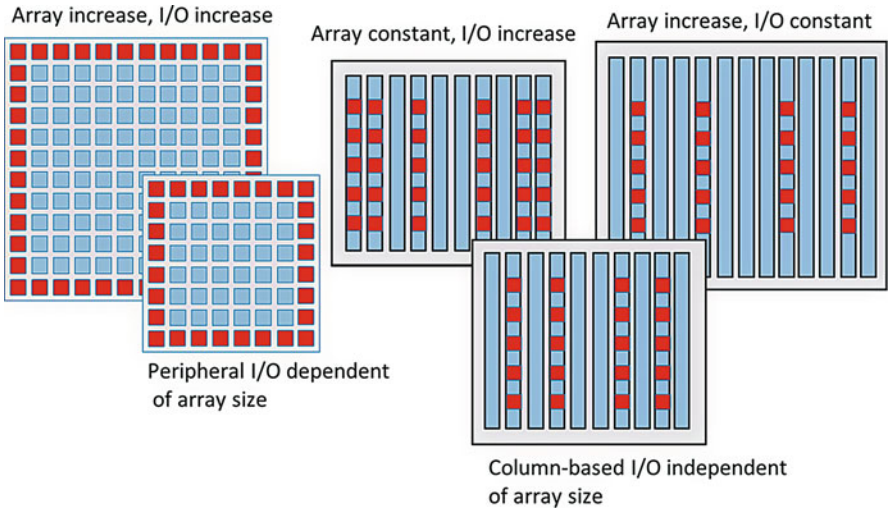


Fig. 11.5 Column-based I/O, enabled by flip-chip packaging technology

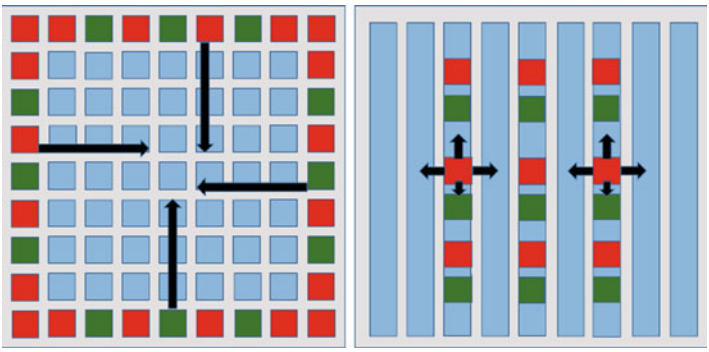


Fig. 11.6 Power and ground distribution in traditional and ASMBL architecture

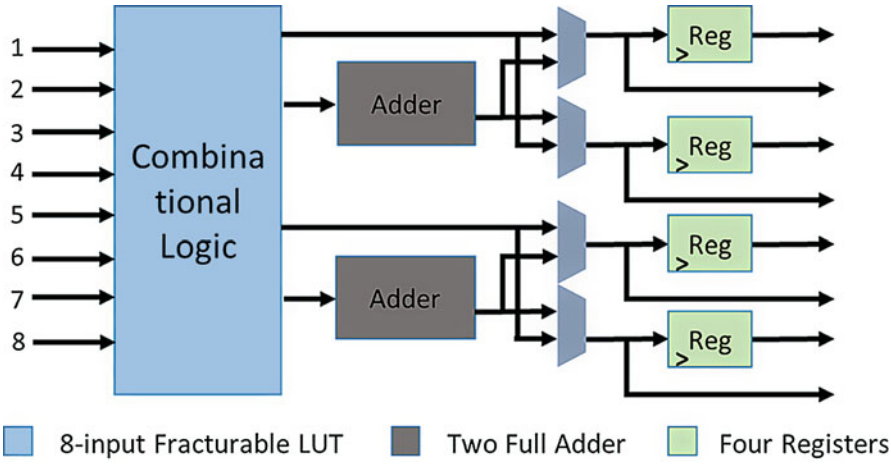


Fig. 11.7 Altera ALM block diagram

structures for data transfer and clock distribution. Stratix FPGAs are based on the MultiTrack interconnect with DirectDrive technology [84]. The MultiTrack interconnect consists of continuous, performance-optimized routing lines of different lengths used for communication within and between distinct design blocks. It also gives more accessibility to any surrounding Logic Array Block (LAB) with much fewer connections, thus improving performance and reducing power. MultiTrack interconnect structure also provides accessing up to 22 clock domains per region. Each Stratix device features up to 16 global clock networks. The DirectDrive technology is a deterministic routing technology, which simplifies the system integration stage of block-based designs by eliminating the often time-consuming system re-optimization process that typically follows design changes and additions.

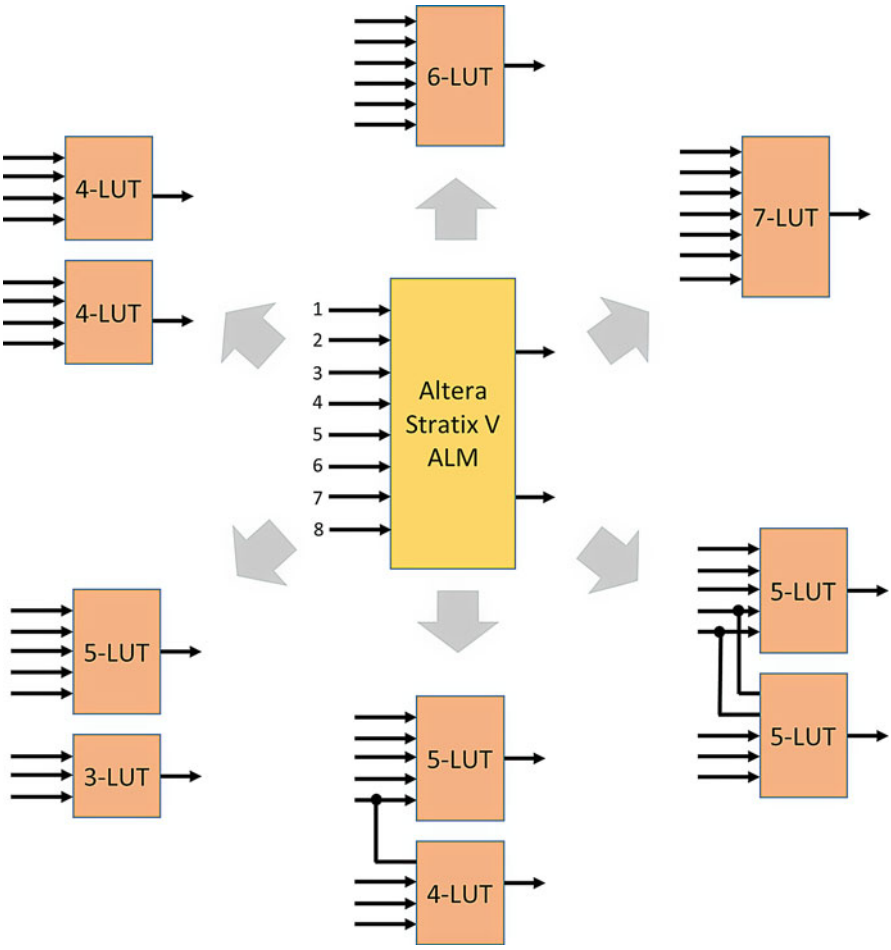


Fig. 11.8 Fracturability of an Altera ALM

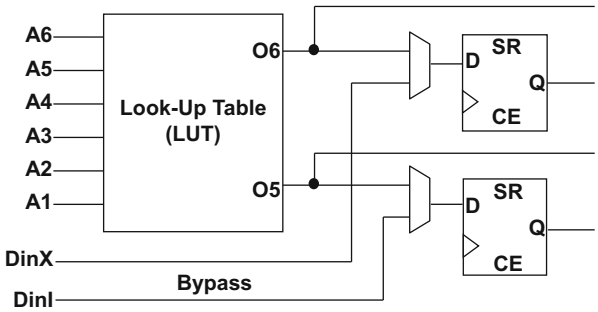


Fig. 11.9 Direct connections to flip-flops bypassing LUT

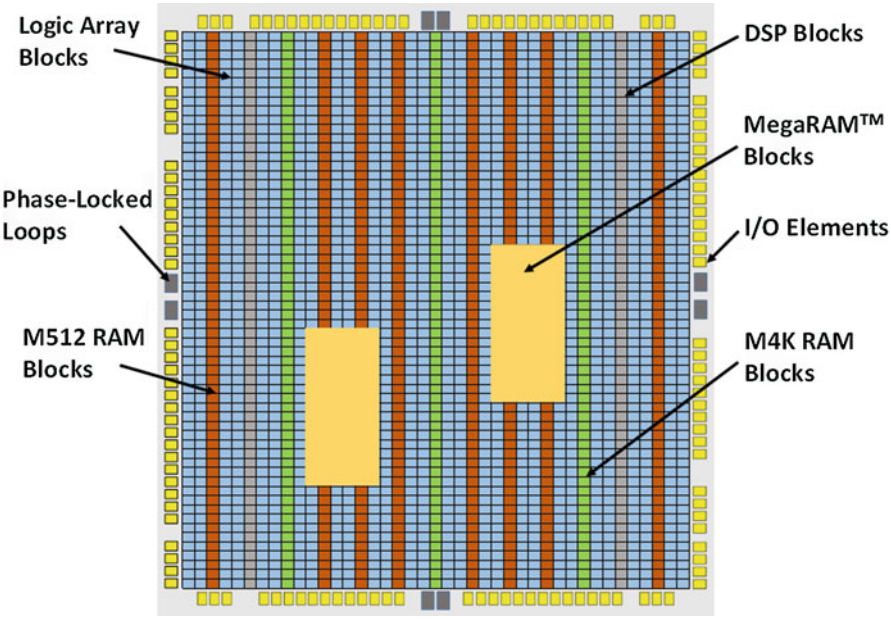


Fig. 11.10 Stratix device architecture

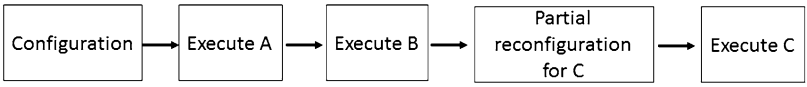


Fig. 11.11 Static partial reconfiguration

11.2.2 Partial Reconfiguration in FPGA

Partial reconfiguration is a feature of modern FPGAs that allows reconfiguration of only a part of the logic fabric of an FPGA. Normally, reconfiguring an FPGA requires it to be held in reset while an external controller reloads a design onto it. Partial reconfiguration allows for critical parts of the design to continue operating while a controller either on the FPGA or off of it loads a partial design into a reconfigurable module. Partial reconfiguration can also be used to save space for multiple designs by only storing the partial designs that change between designs. Partial reconfiguration of FPGAs is a compelling design concept for general purpose reconfigurable systems for its flexibility and extensibility. Partial reconfiguration can be divided into two groups: dynamic partial reconfiguration [68, 97] and static partial reconfiguration.

In static partial reconfiguration, the device is not active during the reconfiguration process. In other words, while the partial data is sent into the FPGA, the rest of the device is stopped and brought up after the configuration is completed, as shown in Fig. 11.11. Dynamic Partial Reconfiguration (DPR), also known as active partial

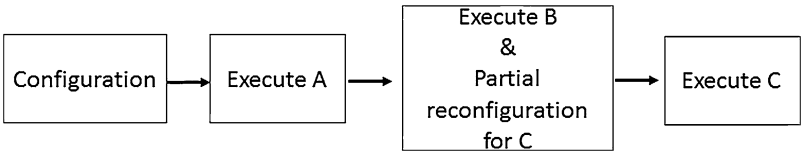


Fig. 11.12 Dynamic partial reconfiguration

reconfiguration, permits to change a part of the device while the rest of an FPGA is still running as illustrated in Fig. 11.12. Nowadays, Xilinx and Altera FPGA vendors support DPR technique in their products [29,69]. The technique can be used to allow the FPGA to adapt to changing hardware algorithms, improve fault tolerance, and achieve better resource utilization. DPR is especially valuable where devices operate in a mission critical environment that cannot be disrupted while some subsystems are being redefined. Placing reconfigurable modules for the partial reconfiguration can be done in different styles such as island, slot, or grid style [28]; depending on the style, a different DPR technique is used. Not all the techniques are supported by FPGA vendors such as Xilinx and Altera, but there are active researches on handling such techniques.

11.2.2.1 Island-Style Reconfiguration

As shown in Fig. 11.13, there are different configuration styles depending on the arrangement of the regions for partial reconfiguration. In the island-style approach, the configurable region is capable of hosting one reconfigurable module exclusively per island. A system might provide multiple islands, but if a module can only run on a specific island, it is called single-island style [54]. If modules can be relocated to different islands, it is called multiple-island style. While the island style can be ideal for systems where only a few modules are swapped, it typically suffers from waste of logic resources due to internal fragmentation in most applications. It happens when modules with different resource requirements exclusively share the same island. For example, if a large module taking a big island is replaced by a smaller one, there will be a waste of logic resources in the reconfigurable region. To alleviate the problem, one can reduce the size of each island, but in that case, a large module may not fit into an island. However, hosting only one module per island makes it simple to determine where to place a module.

11.2.2.2 Slot-Style Reconfiguration

The island-style reconfiguration suffers from a considerable level of internal fragmentation. We can improve this by tiling reconfigurable regions into slots. This results in a one-dimensional slot-style reconfiguration as shown in Fig. 11.13b. In this approach, a module occupies a number of tiles according to its resource requirements, and multiple modules can be hosted simultaneously in a reconfigurable region. Figure 11.13 shows how the tiling influences the spatial packing of modules into a reconfigurable region. In general, however, partial reconfiguration should also consider packing of modules in the time domain. In order to improve the utilization

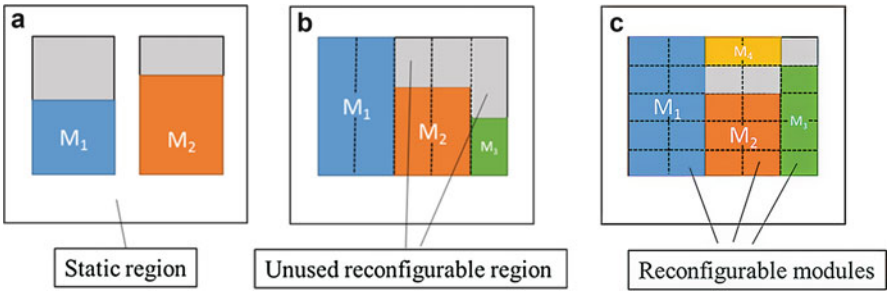


Fig. 11.13 Different placement styles of reconfigurable modules. (**a**) Island style. (**b**) Slot style. (**c**) Grid style

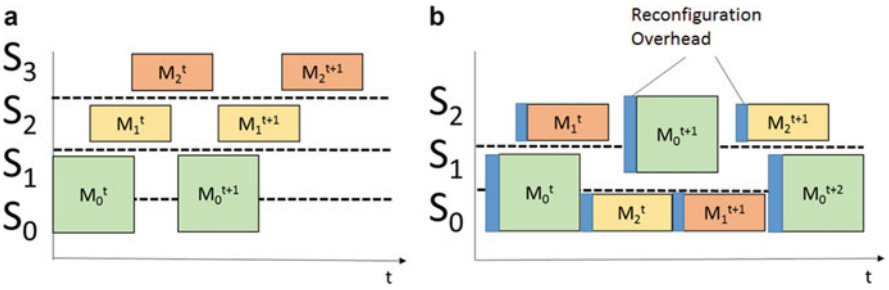


Fig. 11.14 Packing of modules into a system using slot-style reconfiguration. (**a**) Island style. (**b**) Slot style

of the reconfigurable resources by the slot-style reconfiguration [55], modules can be made relocatable to different slots. This is similar to the multiple-island reconfiguration. Figure 11.14 gives an example of how module relocation helps to better fit modules into a reconfigurable region over time. Tiling the reconfigurable region is considerably more complex as the system has to provide communication to and from the reconfigurable modules as well as the placement of the modules. The placement should also consider that FPGA resources are in general heterogeneous. For example, there are different primitives like logic, memory, and arithmetic blocks on the fabric as we mentioned in the last section. Moreover, depending on the present module layout, a tiled reconfigurable region might not provide all free tiles as one contiguous area, which is called external fragmentation. Such an external fragmentation can be removed by defragmenting the module layout which is called compaction.

11.2.2.3 Grid-Style Reconfiguration

The internal fragmentation of a reconfigurable region that is tiled with one dimensional slots can still be large. In particular, the dedicated multiplier and memory resources can be affected much by this since a module typically needs only a few among many resources arranged in columns on the FPGA fabric. Thus

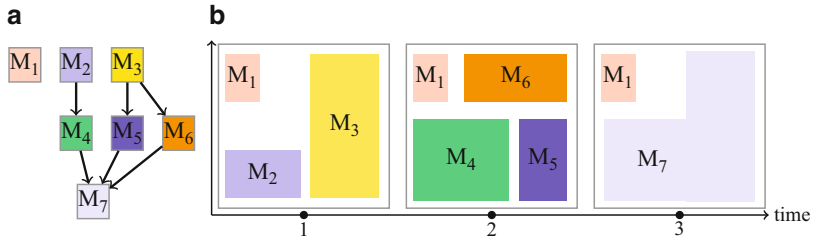


Fig. 11.15 Grid style module packing. (**a**) Task graph. (**b**) Space-time packing

it is beneficial if another module can use the remaining resources by tiling the vertical slots in Fig. 11.13b in horizontal direction. This results in a two-dimensional grid-style reconfiguration [53] as shown in Fig. 11.15b. The implementation and management of such a system is even more complex than the slot-style reconfiguration approach. Note that it requires a communication architecture that can carry out the communication of the modules with the static part of the system and also the communication between reconfigurable modules, and the communication must be established within a system at run time in the absence of sophisticated design tools. Together with the time domain, the two-dimensional grid style placement becomes a three-dimensional packing problem as visualized in Fig. 11.15. The packing should perform scheduling while satisfying the constraints on resource availability and the dependency between the modules. This packing, considering also fragmentation, has to be managed at run time.

11.3 CGRA Architecture

Coarse Grained Reconfigurable Architectures (CGRAs), also known as coarse grained reconfigurable arrays, emerged in 1990s targeting DSP applications [30]. Whereas FPGAs feature bitwise logic in the form of LUTs and switches, CGRAs feature more energy-efficient and area-conscious wordwide PEs, Register Files (RFs), and their interconnections. The cycle-by-cycle reconfigurability of CGRAs along with multiple-bit data-paths has made them superior to FPGAs for repetitive, computation-intensive tasks consisting of various word-level data-processing operations. Wider data-paths in CGRAs allow more efficient implementation of complex operators in silicon. Also the feature of cycle-by-cycle reconfigurability allows customizing the PEs and their connections for every computation and communication, making the performance closer to that of ASIC for word-level operations. Compared to FPGAs, CGRAs have lower delay characteristics and less power consumption. They have much shorter reconfiguration time (cycle level), and thus much more flexible like a programmable processor. On the other hand, gate-level reconfigurability is sacrificed, and thus they are less efficient for bit-level operations.

RaPiD [30], one of the first CGRAs, was developed in 1996. It is a linear array of cells, where each cell comprises of two 16-bit integer ALUs, a 16-bit integer multiplier, six registers, and three small local memories. The interconnections between the functional units are made by segmented buses. It works on 16-bit signed/unsigned fixed-point data. The two 16-bit ALUs in each cell can make a pipelined 32-bit ALU. It runs at 100 MHZ frequency and can perform a sustained rate of 1.6 Giga Operations Per Second (GOPS). Of course, more recent CGRAs can operate at much higher clock frequencies, provide higher power efficiency, and have more PEs in the array [102]. There are plenty of CGRAs designed and implemented in the last two decades; RaPiD, MATRIX [78], Chimaera [41], Raw [110], Garp [42], MorphoSys [100], REMARC [79], CHESS [70], HSRA [106], PipeRench [34], DReAM [8], AVISPA [65], PACT XPP [98], ADRES [73], DAPDNA-2 [99], MORA [58], Chameleon [101], SmartCell [67], FLoRA [62], ReMAP [111], SYSCORE [92], and EGRA [5] are some of THE well-known CGRAs reported to date. A detailed review can be found in survey papers by Hartenstein [39], Todman et al. [104], Choi [20], Tehre et al. [103], and Chattopadhyay [12]. One thing to note is that, in some CGRAs such as ADRES or SRP [51], the architectures work in two modes: CGRA mode and VLIW mode. In such architectures, the VLIW mode executes the control intensive part of the application.

11.3.1 Building Blocks

The main part of CGRA is an array of PEs, RFs, and their interconnections. The array is connected to data and configuration memories as well as a host processor. A PE is basically a unit that performs ALU operations mostly for executing innermost loop kernels. Typically, a PE has its own registers to save temporary data. The host processor may be a VLIW processor (e.g., ADRES), a DSP processor (e.g., Montium), or a general-purpose microprocessor (e.g., MOLEN) to execute non-loop or outer loop code. It also controls the reconfiguration of the array. Data memory works as a communication medium between PE array and the host. Reconfiguration bitstreams reside in the configuration memory and are fed to the array for reconfiguration. The reconfiguration can be done every cycle if the required array behavior changes cycle by cycle. Otherwise, the current configuration can stay in the array for a while without any reconfiguration. Figure 11.16 shows the block diagram of FloRA as a sample CGRA.

11.3.1.1 Processing Elements

CGRAs mostly consist of a 2D (e.g., 8×8) array of cells (PEs) although RaPiD has a linear (1D) array of cells. A cell usually implements a single execution stage but may also include an entire execution unit (RaPiD) or can even be a general-purpose processor (Raw). Figure 11.17a, b show the difference in computing cells between FPGA and CGRA; while a basic cell in FPGA can execute a bit-level operation, the same in CGRA can execute a word-level operation.

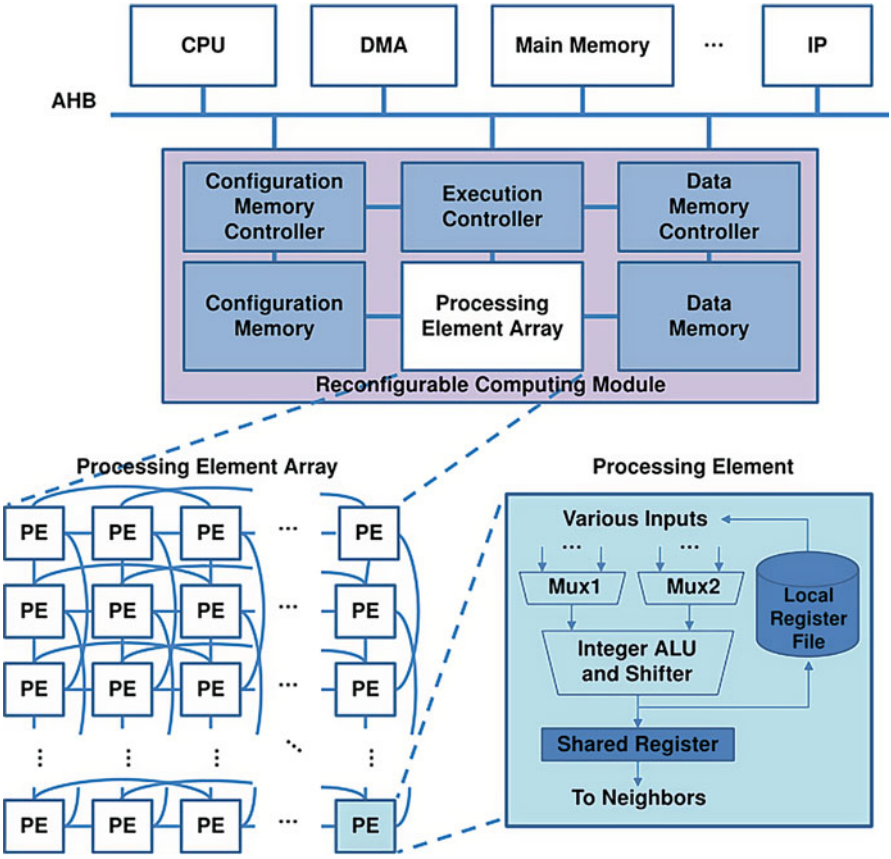


Fig. 11.16 FloRA block diagram

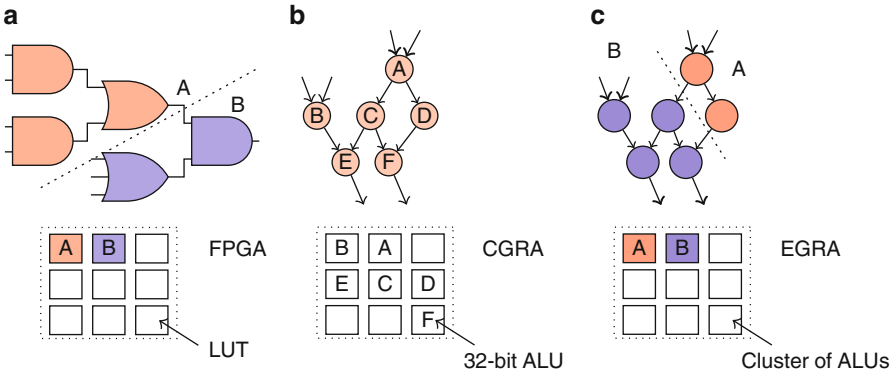


Fig. 11.17 Operation granularity comparison among (**a**) FPGA, (**b**) CGRA and (**c**) EGRA [5]

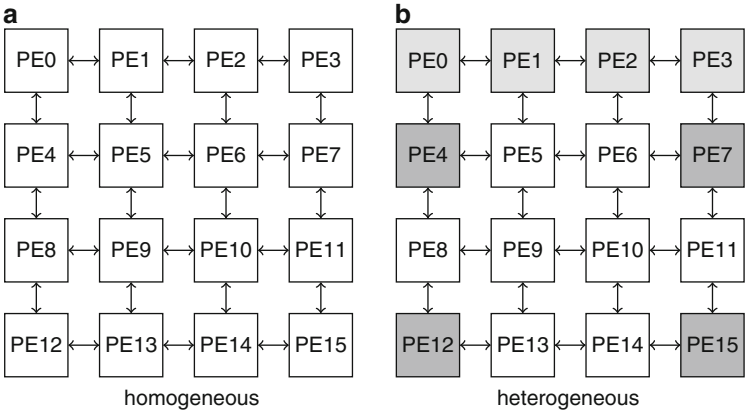


Fig. 11.18 Homogeneous vs. heterogeneous CGRAs

Although some CGRAs have different names for the computing cells, each cell is commonly called a processing element or a PE in short. PEs in different CGRAs support different set of operations (e.g., 16 instructions for CHESS). The number of PEs in a CGRA array varies from 16 in SRP to 64 in FloRA and even more to 24 rows of 16 cells in ReMAP. The PEs are either homogeneous or heterogeneous. While the homogeneity provides a uniform architecture and thus is easier to use, the heterogeneity targets better resource utilization and therefore less power and area consumption. CGRAs such as MORA, MorphoSys, REMARC, and SYSCORE are homogeneous as shown in Fig. 11.18a. Each PE has an ALU, a multiplier, and a register file and is configured through a 16- or 32-bit context word [100]. On the other hand, BilRC, MATRIX, XPP, and SRP are heterogeneous (Fig. 11.18b). Some PEs in SRP support scalar operations while some other support vector operations as well. Besides that, the type and number of operations are not the same among different PEs in SRP. Another aspect of heterogeneity is in accessing the data memory by using read and write operations; only a few PEs, called load/store PEs have access to the data memory [49]. For example, a CGRA array may have one load/store PE per row, while one or two PEs per row may provide multiplications [38]. A heterogeneous architecture may allow normal PEs to share expensive resources (like multipliers), which leads to less area and energy consumption [90]. The sample heterogeneous CGRA illustrated in Fig. 11.18b has three different kinds of PEs in the array. For example, PE0, PE1, PE2, and PE3 can be load/store PEs; PE4, PE7, PE12, and PE15 can contain expensive functional units; other PEs are normal ones.

In architectures like FloRA [38], each PE contains its own RF. But in some other CGRAs like SRP, a register file is shared among a number of PEs. PipeRench, FloRA [44], and SRP are among a few CGRAs that their PEs support floating-point operations besides the integer operations. MorphoSys is a CGRA architecture that works in a SIMD format and is also appropriate for systolic array kind of

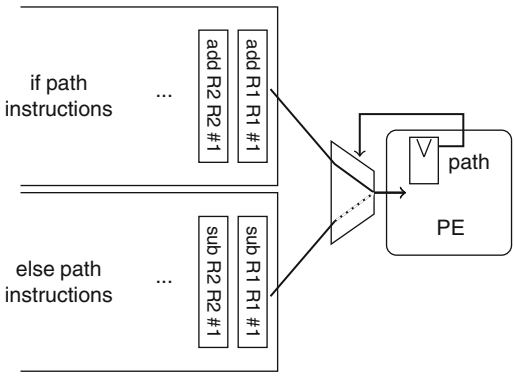


Fig. 11.19 Branch predicate techniques in FloRA

operations. Predicated execution in some new CGRAs such as SRP and FLoRA enables accelerated execution of control flows on a CGRA. Figure 11.19 shows the implementation of predicated executions in FloRA [38].

Raw [110] is another architecture in this category. The main element in the array is called a tile, which contains instruction and data memories, an arithmetic logic unit, registers, configurable logic, and a programmable switch that supports both dynamic and compiler-orchestrated static routing. On the other hand, Garp [42] is something between FPGA and CGRA, having 2-bits wide operations for the logic blocks. Template Expression Grained Reconfigurable Array (EGRA) [5] in Fig. 11.17c is another example of a coarse-grained array. RAC, a complex cell at the heart of the arithmetic in EGRA, supports efficient computation of an entire sub-expression, as opposed to a single operation. In addition, RACs can generate and evaluate branch conditions and be connected either in a combinational or a sequential mode. Figure 11.17c illustrates how a complete expression can be mapped to a cell in EGRA.

The processing element in the reMORPH array is a tile built using DSP and RAM blocks which are already available in an FPGA platform for ALUs and local data/code memories, respectively. Each tile can implement arithmetic and logic operations along with direct and indirect addressing to the data in memory. This enables complete C style loops to be executed on a PE. Memory locations are reused to store the intermediate results. In each iteration, the same set of instructions can be executed by updating the base addresses of the registers to read new data using register indirect addressing. As the reconfiguration of reMORPH array is done at the task level, it is sometimes considered as a many-core architecture rather than a CGRA.

11.3.1.2 Interconnects

In general, CGRAs execute only loops; therefore, they need to be coupled to a host processor. While the array executes the kernel loops, the host processor can execute other parts of the application. Therefore, interconnections in CGRAs can be discussed in two different levels: intra-connections and inter-connections.

Inter-connections define the connections between the array and the host processor, and intra-connections define connections among PEs in the array.

Figure 11.16 shows that a PEs array is connected to the host processor using a common bus, which also connects the PEs array to the main memory. However, the array is connected to its own data memory using direct connections. On the other hand, in some other architectures like ADRES, there is no separate host processor, but part of the array works in a VLIW mode for the role of the host processor.

As the intra-connection, which defines the connections inside a PE array, segmented buses are used among the functional units in RapiD. The most common connection topology in a 2D array of PEs is a mesh connecting a PE to its four nearest neighbors (Fig. 11.20a). Such a mesh is the base interconnection topology in architectures like MorphoSys and ADRES. Figure 11.20 illustrates some other interconnection topologies among PEs including next hop (Fig. 11.20b), buses (Fig. 11.20c), and extra (Fig. 11.20d). Some CGRAs combine mesh interconnects with next-hope connections to provide more routing capabilities among PEs

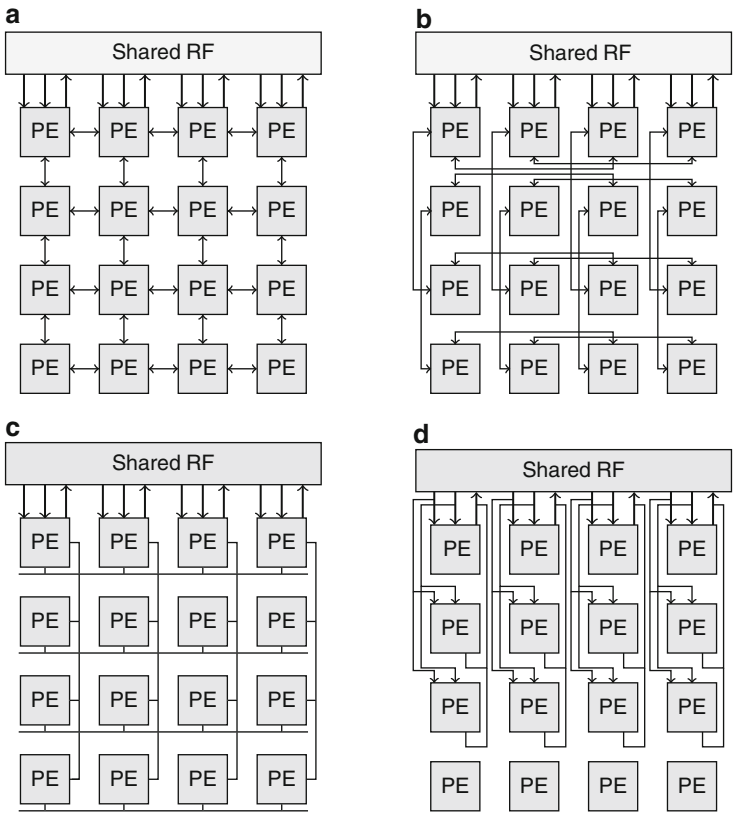


Fig. 11.20 Basic interconnects that can be combined [25]. (**a**) Nearest neighbor. (**b**) Next hop. (**c**) Buses. (**d**) Extra

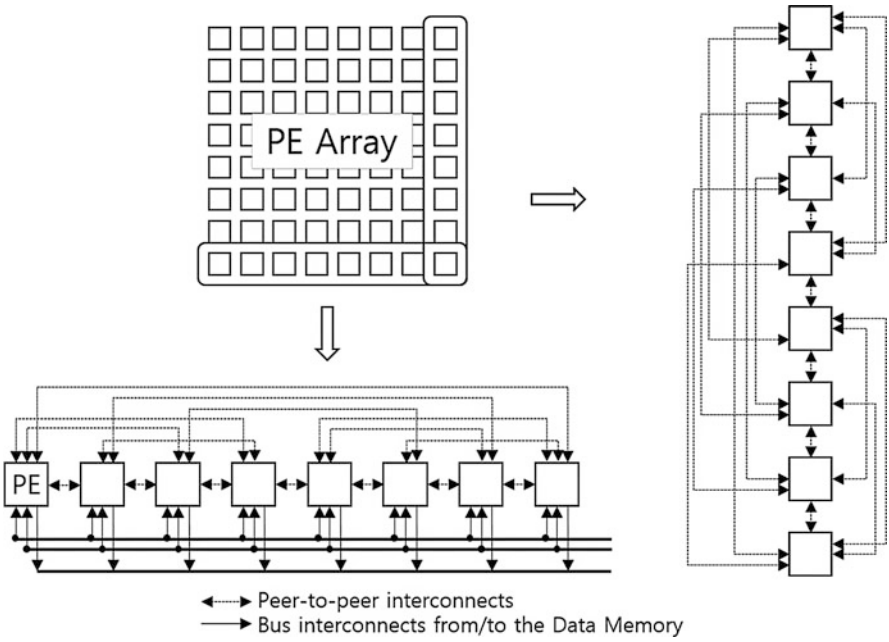


Fig. 11.21 FloRA interconnect network [76]

(e.g., FloRA). Horizontal and/or vertical buses are other common interconnects among PEs in some architectures as shown in Fig. 11.21 [25]. While Fig. 11.20 shows examples of flat interconnection, there are CGRAs with a hierarchical structure supporting multi-level interconnects among PEs. As an example, the PE array in SRP has a 2-level hierarchical interconnect topology. A cluster of PEs form a minicore with a full connection between them. Besides that there is a full connection between minicores [51].

In CGRAs with shared resources and heterogeneous PEs, the connections between PEs and shared resources can follow different topology than the connections between PEs of same or different type. But in most CGRAs, PEs in a row or column share the same resources. In heterogeneous architectures, The load/store PEs may follow the same connection topology as other PEs but they have separate dedicated connections to the ports of the data memory.

11.3.2 Reconfiguration in CGRAs

The reconfigurability of CGRA arrays can be categorized into static reconfiguration, partially dynamic reconfiguration, and fully dynamic reconfiguration.

KressArray is a statically reconfigurable CGRA. In the architecture, the array is configured before a loop is entered. So the mapping is spatial and no reconfiguration

takes place during the loop execution. In such architectures each resource is assigned a single task for executing the loop. Therefore, the associated compiler performs task mapping and data routing, which is similar to the place & route process in FPGA. The spatial mapping in such CGRAs leads to less power consumption, but a large loop cannot be mapped onto the array.

ADRES, Silicon Hive, and MorphoSys support fully dynamic reconfiguration. In such architectures, One full reconfiguration takes place for every execution cycle. Therefore, more than one task can be assigned to a resource during the loop execution lifetime and thus the loop size is not a problem. In this case, the CGRA is treated as a 3D spatial-temporal array, with time (or cycles) as the third dimension. The power consumption of the configuration memories is one drawback for these architectures. SIMD structure of MorphoSys decreases power consumption overhead by fetching one configuration code for all the PEs in a row (or a column). Another technique to reduce power consumption overhead is to pipeline the current configuration of a column to the next column for the next execution cycle [48]. Compressing configuration memory content is another solution to reducing power consumption as well as required memory capacity [86].

PACT and RaPiD feature a partial dynamic reconfiguration, such that part of the configuration bitstream is downloaded to the array statically while the other part is invoked and downloaded onto the array dynamically by using a sequencer. PACT CGRA can initiate events to invoke (partial) reconfiguration [25].

11.4 Mapping onto FPGAs

Traditional mapping of an application onto an FPGA is at the logic level mostly involving technology mapping of logic operations to FPGA logic blocks. As the systems become more complex, however, it is preferred to start the design process at a higher abstraction level such as Electronic System Level (ESL), where high-level programming languages such as C, C++, or SystemC are used to describe the system behavior and then the High-Level Synthesis (HLS) technique is used to automatically generate the Register Transfer Level (RTL) structure that implements the behavior. Figure 11.22 describes the HLS flow.

The compilation, which is the first step of the flow, transforms the input behavioral description into a formal representation. This first step may include various code optimizations such as false data dependency elimination, dead-code elimination, and constant folding. The formal model produced by the compilation exhibits the data and control dependencies between the operations. Data dependencies can be easily represented with a Data-Flow Graph (DFG) in which nodes represent operations and the directed arcs between the nodes represent the input, output, and temporary variables for data dependencies. Such a simple representation does not support branches, loops, and function calls and thus it is extended by adding control dependencies to obtain Control-/Data-Flow Graph (CDFG). There are various ways of combining data flow and control flow into a CDFG. For example, a CDFG can be a hierarchical graph where each node is a DFG that represents a

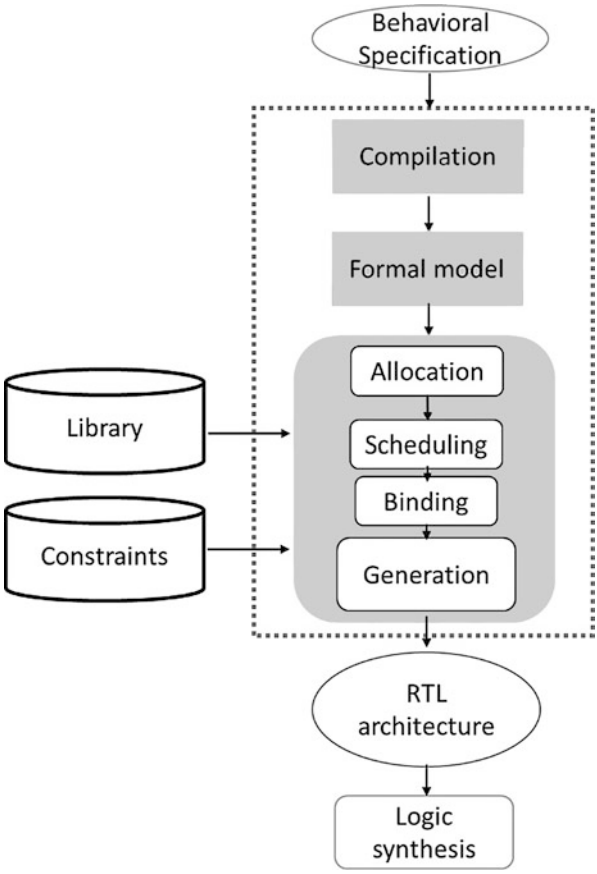


Fig. 11.22 High-level synthesis (HLS) procedures

basic block and edges between the nodes represent control dependencies. Once the CDFG has been built, additional analyses or optimizations can be performed mostly focusing on loop transformations including loop unrolling, loop pipelining, loop fission/fusion, and loop tiling. These techniques are used to optimize the latency or the throughput. To the optimized CDFG, a typical HLS process applies three main steps, namely, allocation, scheduling, and binding. We will discuss those steps in the following subsections. Several HLS tools have been developed for FPGAs targeting specific applications. GAUT is a high-level synthesis tool that is designed for DSP applications [24]. GAUT synthesizes a C program into an architecture with a processing, communication, and memory unit. It requires the user supply specific constraints, such as the pipeline initiation interval. ROCCC is an open-source HLS tool that can create hardware accelerators from C [108]. ROCCC is designed to accelerate kernels that perform repeated computation on streams of data such as FIR filters in DSP applications. ROCCC supports advanced optimizations such as systolic array generation, temporal common subexpression elimination, and it can generate Xilinx PCore modules to be used with a Xilinx MicroBlaze processor [77].

```
void FIR(short*y, short c[N], short x){
  ...
  acc=0;
  Shift_Accum_Loop: for (i=N-1;i>=0;i--){
    if (i==0){
      shift_reg[0]=x;
      data = x;
    }
    else{
      shift_reg[i]=shift_reg[i-1];
      data = shift_reg[i];
    }
    acc+=data*c[i];;
  }
  *y=acc;
}
```

Fig. 11.23 FIR filter C code

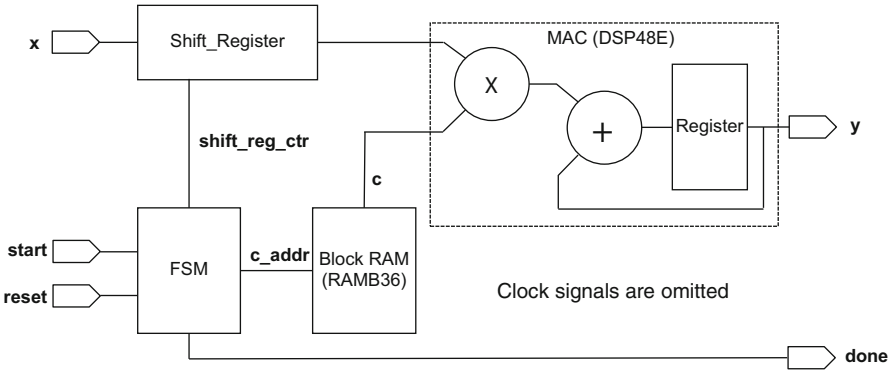


Fig. 11.24 FIR filter block diagram generated by Xilinx Vivado HLS tool

Xilinx developed the Vivado HLS tool [109] based on AutoPilot (a commercial version of xPilot [17]), a product of AutoESL which was acquired by Xilinx. It uses a Low-Level Virtual Machine (LLVM) [59] compilation infrastructure and optimizes various parameters such as interconnect delays, memory configurations, and I/O ports/types for different implementation platforms. It can automatically generate RTL code from an untimed or partially timed C, C++, or SystemC description. Figure 11.23 shows an example of C code for an FIR filter with a 16-bit wide data path, and Fig. 11.24 shows the RTL block diagram generated by the Vivado HLS tool from the C code.

The LegUp [11] is an open-source HLS framework that aims to provide the performance and energy benefits of hardware, while retaining the ease-of-use associated with software. LegUp automatically compiles a standard C program to target a hybrid FPGA-based software/hardware system-on-chip, where some

program segments execute on an FPGA-based 32-bit MIPS soft processor and other program segments are automatically synthesized into FPGA circuits – hardware accelerators – that communicate and work in tandem with the soft processor. LegUp also uses LLVM compiler framework for high-level language parsing and its standard compiler optimizations.

11.4.1 Allocation

Allocation defines the type and the number of hardware resources (functional units, storage, or connectivity components) needed to implement the behavior while satisfying the design constraints. Depending on the HLS tool, some components may be added during scheduling or binding [23]. For example, functional units such as adders or multipliers can be added during scheduling or binding if the given performance constraint cannot be met with the allocated resources. The components are selected from the RTL component library. It is important to select at least one component for each operation type used in the behavioral specification. The library must also include component characteristics such as area, delay, and power consumption.

11.4.2 Scheduling

Scheduling algorithms automatically assign control steps to operations subject to design constraints. These algorithms can be classified into two types: exact algorithms and heuristics. Exact algorithms like the one based on Integer Linear Program (ILP) [33, 43] provide an optimal schedule but take prohibitively long execution time in most practical cases. To cater to the execution time issue, various algorithms based on heuristics have been developed. For example, an algorithm may make a series of local decisions, each time selecting the single best operation-control step pairing without backtracking or look-ahead. So it may miss the globally optimal solution, but can quickly produce a result that is sufficiently close to the optimum and thus acceptable in practice. Examples of basic heuristic algorithms for HLS include As Soon as Possible (ASAP), As Late As Possible (ALAP), List Scheduling (LS), and Force-Directed Scheduling (FDS).

FDS and LS are constructive heuristic algorithms, and the quality of the results may be limited in some cases. To further improve the quality, an iterative method can be applied to the result of constructive method. In [87], for example, they adopt the concept of Kernighan and Lin's heuristic method for solving the graph-bisection problem [45] to reschedule operations into an earlier or later step iteratively until maximum gain is obtained. There are many other iterative algorithms for the resource constrained problem including genetic algorithm [7], tabu search [6, 94], simulated annealing [10, 19], and graph theoretic and computational geometry approaches [3].

11.4.3 Binding

Every operation in the specification or CDFG must be bound to one of the functional units capable of executing the operation. If there are several units with such capability, the binding algorithm must optimize this selection. Each variable that carries values from an operation to another operation across cycles (or control steps) must be bound to a storage unit. In addition, each data transfer from component to component must be bound to a connection unit such as a bus or a multiplexer. Ideally, high-level synthesis estimates the connectivity delay and area as early as possible so that later steps of HLS can better optimize the design. An alternative approach is to specify the complete architecture during allocation so that initial floor planning results can be used during binding and scheduling.

There are many algorithms proposed, but some of the basic ones include clique partitioning, left-edge algorithm, and iterative refinement. In the clique partitioning-based binding [83], the operations and variables are modeled as a graph. Cong and Smith [14] present a bottom-up clustering algorithm based on recursive collapsing of small cliques in a graph. Kurdahi and Parker [56] solved the register binding problem for a scheduled data-flow graph by using the left-edge algorithm. Chen and Cong [18] propose the k-cofamily-based register binding algorithm targeting multiplexer optimization problem.

11.4.4 Technology Mapping

Most modern FPGA devices contain programmable logic blocks that are based on a K-input look-up table (K-LUT) where a K-LUT contains 2^K truth table configuration bits so it can implement any K-input function. Thus, any logic circuit can be implemented with one K-LUT, provided that the circuit has only one output and the number of inputs is not larger than K; the internal complexity of the circuit does not matter.

The number of LUTs needed to implement a given circuit determines the size and cost of the FPGA-based realization. Thus one of the most important phases of the FPGA CAD flow is the *technology mapping* step that maps a circuit description into a LUT network presented in the target FPGA architecture, while minimizing the number of LUTs used for the mapping and the critical path delay. The process of technology mapping is often treated as a covering problem. For example, consider the process of mapping a circuit onto a network of LUTs as illustrated in Fig. 11.25. Figure 11.25a illustrates the original gate-level circuit and a possible covering with three 5-LUTs. Figure 11.25b illustrates a different mapping of the circuit through overlapped covering. In the mapping, the gate labeled X is duplicated and covered by both LUTs. Gate duplication like this example is often necessary to minimize the number of LUTs used for the mapping [22].

There are several methods for technology mapping including graph-based and LUT-based methods. Chen et al. [16] introduce graph-based FPGA technology mapping for delay optimization. As a preprocessing phase of this work, a general

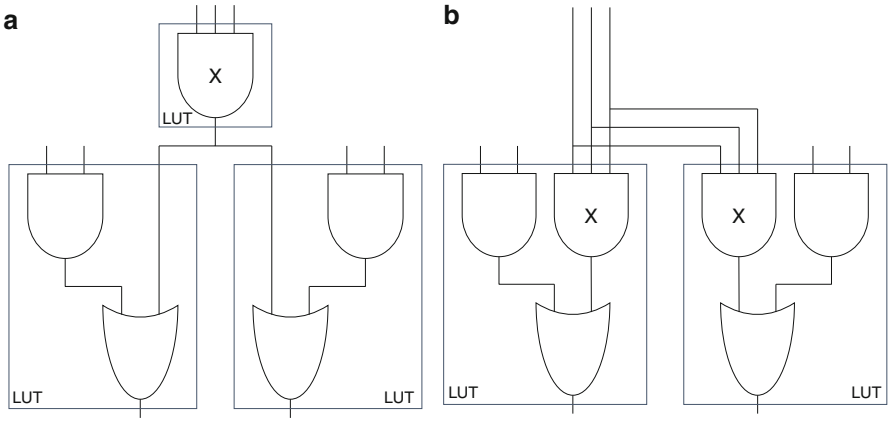


Fig. 11.25 Technology mapping as a covering problem. (**a**) Gate-level circuit and mapping. (**b**) Better mapping with duplication

algorithm called DMIG transforms an arbitrary n-node network into a network consisting of at most two-input gates with only an O(1) factor increase in network depth. A matching-based technique that minimizes area without increasing network delay is used in the post-processing phase. Cong and Minkovich [21] present LUT-based FPGA technology mapping for reliability. As device size shrinks to the nanometer range, FPGAs are increasingly prone to manufacturing defects, and it is important to have the ability to tolerate multiple defects. One common defect point is in the LUT configuration bits, which are crucial to the correct operation of FPGAs. This work presents an error analysis technique that efficiently calculates the number of critical bits needed to implement each LUT. It allows the design to function correctly when implemented on a faulty FPGA.

11.5 Mapping onto CGRAs

Despite the enormous computation power, the performance of CGRAs critically hinges on a smart compiler and mapping algorithm. The target applications of these architectures often spend most of their time executing a few time-critical loop kernels. So the performance of the entire application may be improved considerably by mapping these loop kernels onto an accelerator. Moreover, these computation-intensive loops often exhibit a high degree of inherent parallelism. This makes it possible to use the abundant computation resources available in CGRAs. The programmer or the compiler for a CGRA may find these computation-intensive loops through profiling and/or analysis and directs the computation-intensive segments to CGRA and control-intensive part to the host processor.

The first compilation attempts were focused on ILP but failed to better exploit the parallelism than VLIW [74]. Success of software pipelining techniques encouraged researches to examine modulo scheduling. Modulo scheduling is a software

pipelining technique used in VLIW to improve parallelism by executing different loop iterations in parallel. The objective of modulo scheduling is to engineer a schedule for one iteration of the loop such that the same schedule is repeated at regular intervals with respect to intra- and inter-iteration dependencies and resource constraints. This interval is termed Initiation Interval (II), essentially reflecting the performance of the scheduled loop. It is determined by several parameters, and the reader is directed to [95] for the details.

Modulo scheduling on coarse-grained architectures is a combination of three subproblems: placement, routing, and scheduling. Placement determines on which PE to place one operation. Scheduling determines in which cycle to execute that operation. Routing connects the placed and scheduled operations according to their data dependencies [25, 74]. In the worst case, II is equal to the schedule length (iteration length), and in the best case, it is equal to one, which means that the entire loop is mapped onto the CGRA at once (static mapping). In case of II≥2, PEs need to be reconfigured several times to execute the entire loop.

Dynamically Reconfigurable Embedded System Compiler (DRESC) [74] uses a modulo scheduling algorithm based on simulated annealing [52]. It begins with a random placement of operations on the PEs, which may not be a valid modulo schedule. Operations are then moved between PEs until a valid schedule is achieved. The random movement of operations in the simulated annealing technique can result in a long convergence time for loops with modest numbers of operations [89]. SPR [31] is a mapping tool that uses Iterative Modulo Scheduling (IMS), besides using simulated annealing placement with a cooling schedule inspired by VPR [9] as well as PathFinder [71] and QuickRoute [66] for pipelined routing.

To have a better mapping, it is required to consider scheduling, placement, and routing at the same time. Graph-based algorithms [74, 75, 116, 117] are able to do the job just by modeling CGRA as a graph including time as the third dimension. Therefore, the mapping problem becomes mapping the loop kernel DFG onto the CGRA Modulo Resource Routing Graph (MRRG). Figure 11.26 shows how a loop kernel DFG is mapped onto the CGRA, where three subsequent iterations of the DFG are mapped. As shown in this example, II is 2 while the schedule length is 4.

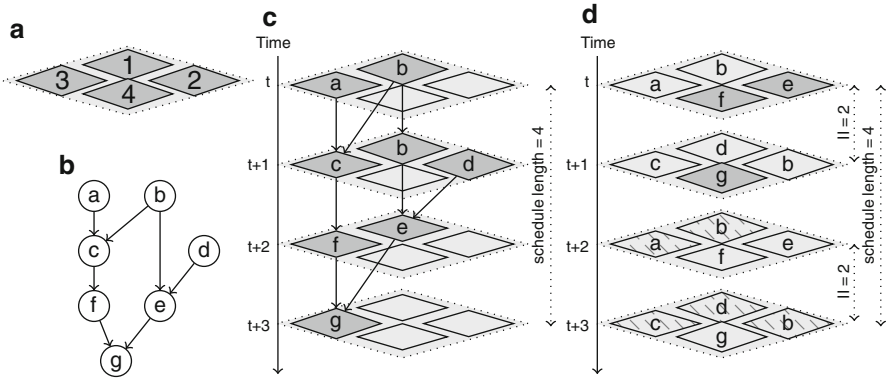


Fig. 11.26 Mapping example [35]

Figure 11.26d shows that the same PE which is doing operation "b" in cycle "i," acts as routing PE in cycle "i+1," to route operation "b" to cycle "i+2." In this way, assigning some PEs for routing provides more connectivity for the communications between nodes of a DFG.

One of the parameters defining II is the recurrence-constrained lower bound. Oh et al. [85] introduced a recurrence cycle-aware scheduling technique for CGRAs. Their modulo scheduler groups operations belonging to a recurrence cycle into a clustered node and then computes a scheduling order for those clustered nodes. Deadlocks that arise when two or more recurrence cycles depend on each other are resolved by using heuristics that favor recurrence cycles with long recurrence delays. Whereas previous approaches had to sacrifice either compilation speed or quality of the result, this is no longer necessary with the recurrence cycle-aware scheduling technique. Traditional schedulers are node-centric in that the focus is assigning operations to PEs. The straightforward adaptation of this approach is operation placement followed by operand routing to determine if the assignment is feasible. Park et al. [89] have shown that node-centric approaches are poor for CGRA. They proposed an Edge Centric Modulo Scheduling (EMS) approach. This approach focuses on mapping edges instead of nodes.

Shared resources [47], data memory limitation [26, 49], and register file distribution (REGIMap [36]) are also important constraints that must be considered for the mapping of a DFG onto a CGRA. ILP can be used to obtain an optimal solution to a mapping problem considering such constraints. However, since the ILP approach is slow in general, it is used to obtain an optimal solution for problems of small size; the solution is used to check the quality of other heuristic-based (non-ILP) approaches. We briefly introduce some of the ILP-based and heuristic-based mapping approaches in Sects. 11.5.1 and 11.5.2, respectively.

In cases that there is not enough space to map all the loop kernels of the application, some decision has to be made to find more eligible kernels. Lee et al. [64] have proposed a kernel selection algorithm. If the memory requirement of the application is larger than the available Scratchpad Memory (SPM) size, kernel selection is performed based on detailed statistics such as run-time and buffer-access information of each kernel. Otherwise, all the kernels are mapped to the CGRA.

11.5.1 ILP-Based Mapping Approaches

There have been a few approaches to ILP formulation of the problem of mapping an application to a CGRA. Ahn et al. [1] have formulated the mapping problem in ILP for the first time. Their approach consists of three stages: covering, partitioning, and laying-out. In the covering stage, a kernel tree is transformed to the configuration tree such that each node of the configuration tree represents a configuration for each PE and can cover and execute one or more operations. The partitioning stage splits the configuration tree to clusters such that each cluster is mapped to one distinct column of the CGRA in laying-out stage. The authors have targeted optimal vertical mapping with the minimum total data transfer cost among the rows of PEs.

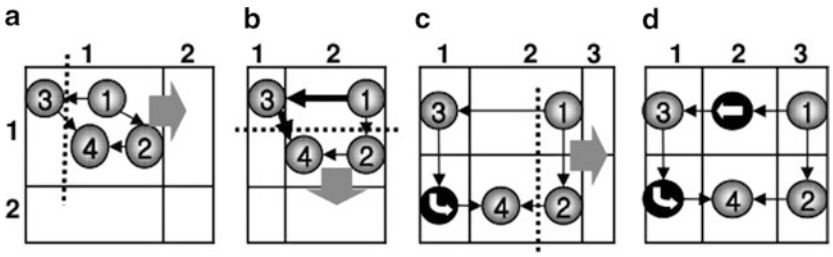


Fig. 11.27 Split & Push heuristic in SPKM [116]

Yoon et al. [117] have developed a graph-based ILP formulation. Then they have used Split & Push Kernel Mapping (SPKM) heuristic to solve the mapping problem within a feasible time. Figure 11.27 shows how the Split & Push Kernel Mapping Algorithm works. It assigns the entire DFG into one PE and then starts splitting it horizontally and vertically. The link between the non-neighboring PEs is fulfilled by using routing PEs. Their formulation takes into account many architectural details of CGRA and leads to minimum number of rows.

Lee et al. [63] have proposed an approach that covers not only integer operations but also floating-point operations implemented by simply using two neighboring tiles. Besides their ILP formulation, they have developed a fast heuristic mapping algorithm considering Steiner points. Details are given in Sect. 11.5.3.

As already mentioned, there are CGRAs like Raw [110] and reMORPH [80, 93] that provide reconfigurations at the task level rather than instruction level. Moghaddam et al. [81, 82] have presented an ILP-based optimal framework to map an application in the form of a task graph onto a tile-based CGRA. They have integrated scheduling, placement, and routing into one mapping problem. The formulation benefits from the reconfigurability feature of the target platform; a large application having more tasks than the number of PEs or even multiple applications can be mapped to the platform.

11.5.2 Heuristic-Based Approaches

There are many heuristic-based mapping approaches for CGRAs including EMS [89], EPIMap [35], and graph-minor approach [15]. We review some of the most referenced ones here.

Lee et al. [60] have developed a generic architecture template, called the dynamically reconfigurable ALU array Dynamically Reconfigurable ALU Array (DRAA). Their mapping approach goes through the following three levels: PE level, line level, and plane level. In the PE level, a DFG is extracted. In the line level, nodes of the DFG are grouped such that each group can be assigned to a distinct row of PEs. And finally in the plane level, the lines are stitched together to form a plane. They take into account the data reuse patterns in loops of DSP algorithms as part of their approach.

Park et al. [88] have presented their modulo graph embedding. Modulo graph embedding is also a modulo scheduling technique for software pipelining. They have modelled the architecture using an MRRG. Their MRRG has only II layers, which makes the problem space smaller, and therefore the mapping algorithm converges to the solution faster. They have later [89] presented an EMS approach, which specifically targets routing of data instead of placement of operations.

Galanis et al. [32] have presented a priority-based mapping algorithm. This algorithm assigns an initial priority to each operation of the DFG. This priority is inversely proportional to the mobility, which is the difference between ALAP and ASAP schedule times. The operations residing on the critical path will be scheduled first.

Hanataka et al. [40] have presented a modulo scheduling algorithm that takes into account "resource reservation" and "scheduling" separately. They have used a resource usage aware relocation algorithm. Their approach uses a compact 3D architecture graph similar to the MRRG used in [88]. This graph is only II times as large as the original two-dimensional graph.

Dimitroulakos et al. [27] have presented an efficient mapping approach where scheduling and register allocation phases are performed in one single step. They have also incorporated modulo scheduling with back tracking in their approach. Their mapping approach minimizes memory bandwidth bottleneck. They have tried to maximize the ILP using a new priority scheme and few heuristics. Their solution covers a large range of CGRAs. They have also developed a simulation framework.

Oh et al. [85] have proposed a scheduling technique that is aware of data dependencies caused by inter-iteration recurrence cycles. Therefore, operations in a recurrence cycle are clustered and considered as a single node. The operations in a recurrence cycle are handled as soon as all predecessors of the clustered node have been scheduled. They have also proposed a modification in the target architecture to further improve the quality of their scheduling approach.

Lee et al. [61] have proposed a mapping approach based on high-level synthesis techniques. They have used loop unrolling and pipelining techniques to generate loop parallelized code to improve the performance drastically.

Patel et al. [91] benefit from systolic mapping techniques in their scheduler. They prepare an Synchronous Data Flow (SDF) graph for the application; they rearrange the graph for systolic mapping, schedule the SDF graph, and then prepare a CDFG for each node of the SDF graph. As the last step, they generate topology matrix and delay matrix which are used for the final systolic mapping.

Kim et al. [50] have proposed a memory-aware mapping technique for the first time. They have also proposed efficient methods to handle dependent data on a double-buffering local memory, which is necessary for recurrent loops.

11.5.3 FloRA Compilation Flow: Case Study

FloRA consists of a Reconfigurable Computing Module (RCM) for executing loop kernel code segments and a general-purpose processor for controlling the RCM, and

these units are connected with a shared bus. The RCM consists of an array of PEs, several sets of data memories, and a configuration memory [47]. Figure 11.16 shows FloRA containing a 8×8 reconfigurable array of PEs and internal structure of a PE. Each PE is connected to the nearest neighboring PEs: top, bottom, left, and right. The size of the array can be optimized to a specific application domain.

The area-critical resources (such as multipliers) are located outside the PEs and shared among a set of PEs. Each area-critical resource is pipelined to curtail the critical path delay, and its execution is initiated by scheduling the area-critical operation on one of the PEs that share this area-critical resource. Thus, each PE can be dynamically reconfigured either to perform arithmetic and logical operations with its own Arithmetic-Logic Unit (ALU) in one clock cycle or to perform multiply or division operations using the shared functional unit in several clock cycles with pipelining. Resource pipelining further improves loop pipelining execution by allowing multiple operations to execute simultaneously on one pipelined resource. Furthermore, pipelining together with resource sharing increases the utilization of these area-critical units. Data memory consists of three banks: one connected to the write bus and the other two connected to the read buses. The connections can also be reconfigured. Each PE has its local Configuration Cache Element (CCE). Each CCE has several layers, so the corresponding PE can be reconfigured independently with different contexts.

FloRA supports floating-point operations by allotting a pair of PEs: one for mantissa and the other for exponent. Mapping a floating-point operation onto the PE array with integer operations may take many layers of cache. If a kernel consists of a multitude of floating-point operations, then mapping it onto the array easily runs out of the cache layers, causing costly fetch of additional context words from the main memory. Instead of using multiple cache layers to perform such a complex operation, some control logic is added to the PEs so that the operation can be completed in multiple cycles but without requiring multiple cache layers. The control logic can be implemented with a small Finite-State Machine (FSM) that controls the PE's existing data path for a fixed number of cycles [44].

Lee et al. have presented two mapping approaches for FloRA: (1) an optimal approach using ILP and (2) a fast heuristic approach using Quantum-inspired Evolutionary Algorithm (QEA). Both approaches support integer-type applications as well as floating-point-type applications. These mapping algorithms adopt HLS techniques that handle loop-level parallelism by applying loop unrolling and loop pipelining techniques. The overall compilation flow is given in Fig. 11.28. The first step is partitioning, which generates two C codes one for the RISC processor and the other for the CGRA.

The code segments for the RISC processor are statically scheduled and the corresponding assembly code is generated with a conventional compiler. The code segments for the RCM (generally loop kernels) are converted to a CDFG using the SUIF2 [107] parser. During this process, loop unrolling maximizes the utilization of the PEs. Then HLS techniques are used for the scheduling and binding on one column of PEs. Each column of the CGRA executes its own iteration of the loop to implement loop pipelining.

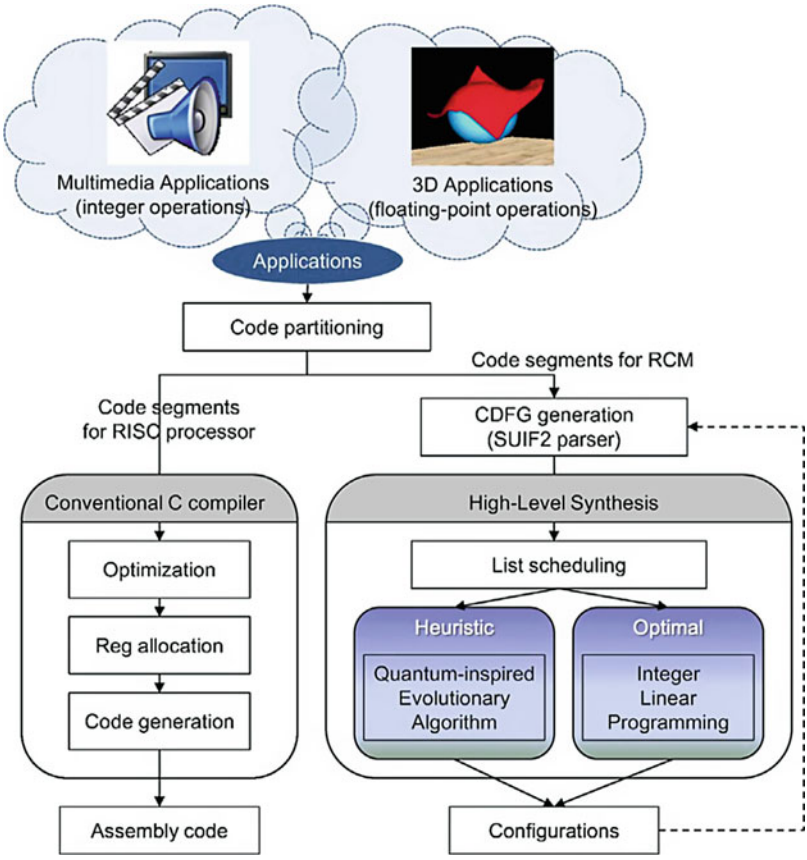


Fig. 11.28 Overall design flow for application mapping onto FloRA [63]

The main objective of the mapping problem is to map a given loop kernel to the CGRA such that the total latency is minimized while satisfying several constraints. Lee et al. have formulated the problem using ILP. ILP-based application mapping yields an optimal solution. However, it takes an unreasonably long execution time to find a solution, making it unsuitable for large designs or for design space exploration. Therefore, they defined a fast heuristic mapping algorithm considering Steiner points. Their heuristic is based on a mixture of two algorithms: List Scheduling and QEA.

11.5.3.1 List Scheduling

First, List Scheduling algorithm topologically sorts the vertices from the sink to the source. If a vertex has a longer path to the sink, then it gets a higher priority. From the sorted list, the algorithm selects and schedules the vertex with the highest priority if all the predecessor vertices have been scheduled and the selected vertex is

reachable from all the scheduled predecessor vertices through the interconnections available in the CGRA. If the vertex is a floating-point vertex, the algorithm checks to see if neighbor PEs are busy, since executing a floating-point operation requires a pair of PEs for several cycles. Mapping a vertex onto a PE considers interconnect constraint and shared resource constraint. If there is no direct connection available for implementing a data dependency between two PEs, a shortest path consisting of unused PEs which work as routers is searched. Another constraint to be considered is the constraint set by sharing area-critical functional units. For example, if there is only one multiplier shared among the PEs in a row, two multiply operations cannot be scheduled successively but should wait for N (number of PEs in a row) cycles after scheduling one multiply operation, since the multiplier must be used by other PEs in the same row for loop pipelining. In this case, the second multiply operation may need to wait with proper routing of the input data.

11.5.3.2 QEA

QEA is an evolutionary algorithm that is known to be very efficient compared to other evolutionary algorithms [37]. The QEA starts from the List Scheduling result as a seed and attempts to further reduce the total latency. Starting the QEA with a relatively good initial solution tends to reach a better solution sooner than starting it with a random seed. When the schedule and binding of all vertices are determined, it tries to find the routing paths among the vertices – the routing may need to use unused remaining PEs – to see if these schedule and binding results violate the interconnect constraint. In this routing phase, the quality of the result depends on the order of edges to be routed. Thus the priority of edges for the ordering is determined as follows.

- Edges located in the critical path are assigned higher priority.
- Among the edges located in the critical path, edges that have smaller slack (shorter distance) receive higher priority.
- If a set of edges have the same tail vertex, then the set of edges becomes a group and the priority of this group is determined by the highest priority among the group members.

According to the above priority, a list of candidate edges is made and a shortest path for each edge is found in the order of priority with the Dijkstra's algorithm. In this routing phase, a Steiner tree (instead of a spanning tree) for multiple writes from a single source is considered. The heuristic algorithm for finding a Steiner tree tries to find a path individually for each outgoing edge from the source. If some paths use the same routing PE, it becomes a Steiner point. Although this approach may not always find an optimal path, it gives good solutions in most of the cases if not all. Indeed, experimental results show that the approach finds optimal solutions for 97% of the randomly generated examples. Table 11.1 compares the result obtained by the heuristic algorithm for the butterfly addition example with the optimum result obtained by the ILP formulation.

Table 11.1 Experimental result of butterfly addition example

		Latency (cycle)	Mapping time (s)
ILP	Spanning tree	5	1022
	Steiner tree	4	965
Heuristic	Spanning tree	5	13
	Steiner tree	4	9

11.6 Conclusions

Reconfigurable architecture provides software-like flexibility as well as hardware-like performance. Depending on the granularity of configuration, we can consider two types of reconfigurable architecture: fine-grained reconfigurable architecture like FPGA and CGRA. In this chapter, we have surveyed various architectures for FPGAs and CGRAs. We have also surveyed various approaches to mapping applications to the architectures. Compared to pure hardware design or pure software design, there are more opportunities in utilizing such reconfigurable architectures since they support hardware reconfiguration which is controlled by software (For general trade-offs between hardware and software, refer to ▸ Chap. 1, "Introduction to Hardware/Software Codesign".). For example, FPGAs can be better utilized by dynamic partial reconfiguration, which has been mentioned in this chapter. However, such opportunities have not been very well investigated and still require more researches together with the researches on better architectural supports.

References

1. Ahn M, Yoon J, Paek Y, Kim Y, Kiemb M, Choi K (2006) A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In: Proceedings of the design, automation and test in Europe, DATE '06, vol 1, p 6
2. Altera arria 10 FPGA. www.altera.com. Accessed 28 Nov 2015
3. Aletà A, Codina JM, Sánchez J, González A (2001) Graph-partitioning based instruction scheduling for clustered processors. In: Proceedings of the 34th annual ACM/IEEE international symposium on microarchitecture, MICRO 34. IEEE Computer Society, Washington, DC, pp 150–159
4. Altera stratix v FPGA. www.altera.com. Accessed 28 Nov 2015
5. Ansaloni G, Bonzini P, Pozzi L (2011) EGRA: a coarse grained reconfigurable architectural template. IEEE Trans Very Large Scale Integr (VLSI) Syst 19(6):1062–1074
6. Baar T, Brucker P, Knust S (1999) Tabu search algorithms and lower bounds for the resource-constrained project scheduling problem. In: Voss S, Martello S, Osman I, Roucairol C (eds) Meta-heuristics. Springer US, pp 1–18. doi: 10.1007/978-1-4615-5775-3_1
7. Bean JC (1994) Genetic algorithms and random keys for sequencing and optimization. ORSA J Comput 6(2):154–160. doi: 10.1287/ijoc.6.2.154, http://dx.doi.org/10.1287/ijoc.6.2.154
8. Becker J, Glesner M (2000) Fast communication mechanisms in coarse-grained dynamically reconfigurable array architecture. In: The 2000 international conference on parallel and distributed processing techniques and applications (PDPTA'2000), Las Vegas

9. Betz V, Rose J (1997) VPR: a new packing, placement and routing tool for FPGA research. In: Proceedings of the 7th international workshop on field-programmable logic and applications, FPL '97. Springer, London, pp 213–222
10. Bouleimen K, Lecocq H (2003) A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. Eur J Oper Res 149(2):268–281. doi: 10.1016/S0377-2217(02)00761-0. Sequencing and Scheduling
11. Canis A, Choi J, Aldham M, Zhang V, Kammoona A, Anderson JH, Brown S, Czajkowski T (2011) LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In: Proceedings of the 19th ACM/SIGDA international symposium on field programmable gate arrays, FPGA '11. ACM, New York, pp 33–36. doi: 10.1145/1950413.1950423
12. Chattopadhyay A (2013) Ingredients of adaptability: a survey of reconfigurable processors. VLSI Des 2013:10:10–10:10
13. Che S, Li J, Sheaffer J, Skadron K, Lach J (2008) Accelerating compute-intensive applications with GPUs and FPGAs. In: Proceedings of the symposium on application specific processors, SASP 2008, pp 101–107
14. Chen D, Cong J (2004) Register binding and port assignment for multiplexer optimization. In: Proceedings of the 2004 Asia and South Pacific design automation conference, ASP-DAC '04. IEEE Press, Piscataway, pp 68–73
15. Chen L, Mitra T (2014) Graph minor approach for application mapping on CGRAs. ACM Trans Reconfig Technol Syst 7(3):21:1–21:25
16. Chen KC, Cong J, Ding Y, Kahng A, Trajmar P (1992) Dag-map: graph-based FPGA technology mapping for delay optimization. IEEE Des Test Comput 9(3):7–20. doi: 10.1109/54.156154
17. Chen D, Cong J, Fan Y, Han G, Jiang W, Zhang Z (2005) xPilot: a platform-based behavioral synthesis system. SRC TechCon 5
18. Chen D, Cong J, Fan Y, Wan L (2010) LOPASS: a low-power architectural synthesis system for FPGAs With interconnect estimation and optimization. IEEE Trans VLSI Syst 18(4): 564–577
19. Cho JH, Kim YD (1997) A simulated annealing algorithm for resource constrained project scheduling problems. J Oper Res Soc 48(7):736–744
20. Choi K (2011) Coarse-grained reconfigurable array: architecture and application mapping. IPSJ Trans SystLSI Des Methodol 4:31–46. doi: 10.2197/ipsjtsldm.4.31
21. Cong J, Minkovich K (2010) Lut-based FPGA technology mapping for reliability. In: Proceedings of the 47th design automation conference, DAC '10. ACM, New York, pp 517–522. doi: 10.1145/1837274.1837401
22. Cong J, Wu C, Ding Y (1999) Cut ranking and pruning: enabling a general and efficient FPGA mapping solution. In: Proceedings of the 1999 ACM/SIGDA seventh international symposium on field programmable gate arrays, FPGA '99. ACM, New York, pp 29–35. doi: 10.1145/296399.296425
23. Coussy P, Gajski D, Meredith M, Takach A: An introduction to high-level synthesis. IEEE Des Test Comput 26(4):8–17 (2009). doi: 10.1109/MDT.2009.69
24. Coussy P, Lhairech-Lebreton G, Heller D, Martin E (2010) Gaut–a free and open source high-level synthesis tool. In: IEEE DATE
25. De Sutter B, Raghavan P, Lambrechts A (2010) Coarse-grained reconfigurable array architectures. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of signal processing systems. Springer, Boston, pp 449–484. doi: 10.1007/978-1-4419-6345-1_17
26. Dimitroulakos G, Galanis MD, Goutis CE (2005) Alleviating the data memory bandwidth bottleneck in coarse-grained reconfigurable arrays. In: 2005 IEEE international conference on application-specific systems, architecture processors (ASAP'05), pp 161–168. doi: 10.1109/ASAP.2005.12
27. Dimitroulakos G, Georgiopoulos S, Galanis MD, Goutis CE (2009) Resource aware mapping on coarse grained reconfigurable arrays. Microprocess Microsyst 33(2):91–105
28. Dirk K (2012) Partial reconfiguration on FPGAs: architectures, tools and applications. Springer, New York

29. Dynamic reconfiguration in Stratix IV devices (2014). Accessed 27 Nov 2015
30. Ebeling C, Cronquist D, Franklin P (1996) Rapid – reconfigurable pipelined datapath. In: Hartenstein R, Glesner M (eds) Field-programmable logic smart applications, new paradigms and compilers. Lecture notes in computer science, vol 1142. Springer, Berlin/Heidelberg, pp 126–135
31. Friedman S, Carroll A, Van Essen B, Ylvisaker B, Ebeling C, Hauck S (2009) SPR: an architecture-adaptive cgra mapping tool. In: Proceedings of the ACM/SIGDA international symposium on field programmable gate arrays, FPGA '09. ACM, New York, pp 191–200. doi: 10.1145/1508128.1508158
32. Galanis M, Dimitroulakos G, Goutis C (2006) Mapping DSP applications on processor/coarse-grain reconfigurable array architectures. In: Proceedings 2006 IEEE international symposium on circuits and systems, ISCAS 2006, p. 4
33. Garfinkel RS, Nemhauser GL (1972) Integer programming, vol 4. Wiley, New York
34. Goldstein S, Schmit H, Budiu M, Cadambi S, Moe M, Taylor R (2000) Piperench: a reconfigurable architecture and compiler. Computer 33(4):70–77
35. Hamzeh M, Shrivastava A, Vrudhula S (2012) EPIMap: using epimorphism to map applications on CGRAs. In: Proceedings of the 49th annual design automation conference, DAC '12. ACM, New York, pp 1284–1291
36. Hamzeh M, Shrivastava A, Vrudhula S (2013) REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In: 2013 50th ACM/EDAC/IEEE design automation conference (DAC), pp 1–10. doi: 10.1145/2463209.2488756
37. Han KH, Kim JH (2004) Quantum-inspired evolutionary algorithms with a new termination criterion, H/sub/spl epsi//gate, and two-phase scheme. IEEE Trans Evol Comput 8(2):156–169. doi: 10.1109/TEVC.2004.823467
38. Han K, Ahn J, Choi K (2013) Power-efficient predication techniques for acceleration of control flow execution on CGRA. ACM Trans Archit Code Optim 10(2):8:1–8:25. doi: 10.1145/2459316.2459319
39. Hartenstein R (2001) A decade of reconfigurable computing: a visionary retrospective. In: Proceedings of the design, automation and test in Europe, Conference and Exhibition 2001, pp 642–649
40. Hatanaka A, Bagherzadeh N (2007) A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In: IEEE international parallel and distributed processing symposium, IPDPS 2007, pp 1–8
41. Hauck S, Fry T, Hosler M, Kao J (2004) The chimaera reconfigurable functional unit. IEEE Trans Very Large Scale Integr (VLSI) Syst 12(2):206–217
42. Hauser J, Wawrzynek J (1997) Garp: a mips processor with a reconfigurable coprocessor. In: Proceedings of the 5th annual IEEE symposium on field-programmable custom computing machines, 1997, pp 12–21
43. Hwang CT, Lee JH, Hsu YC (1991) A formal approach to the scheduling problem in high level synthesis. IEEE Trans Comput-Aided Des Integr Circuits Syst 10(4):464–475. doi: 10.1109/43.75629
44. Jo M, Lee D, Han K, Choi K (2014) Design of a coarse-grained reconfigurable architecture with floating-point support and comparative study. Integr {VLSI} J 47(2):232–241. doi: 10.1016/j.vlsi.2013.08.003
45. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49:291–307
46. Kestur S, Davis J, Williams O (2010) Blas comparison on FPGA, CPU and GPU. In: 2010 IEEE computer society annual symposium on VLSI (ISVLSI), pp 288–293. doi: 10.1109/ISVLSI.2010.84
47. Kim Y, Kiemb M, Park C, Jung J, Choi K (2005) Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In: Design, automation and test in Europe, vol 1 pp 12–17 . doi: 10.1109/DATE.2005.260
48. Kim Y, Mahapatra RN, Park I, Choi K (2009) Low power reconfiguration technique for coarse-grained reconfigurable architecture. IEEE Trans Very Large Scale Integr (VLSI) Syst 17(5):593–603. doi: 10.1109/TVLSI.2008.2006039

49. Kim Y, Lee J, Shrivastava A, Paek Y (2011) Memory access optimization in compilation for coarse-grained reconfigurable architectures. ACM Trans Des Autom Electron Syst 16(4):42:1–42:27. doi: 10.1145/2003695.2003702
50. Kim Y, Lee J, Shrivastava A, Yoon J, Cho D, Paek Y (2011) High throughput data mapping for coarse-grained reconfigurable architectures. IEEE Trans Comput-Aided Des Integr Circuits Syst 30(11):1599–1609
51. Kim C, Chung M, Cho Y, Konijnenburg M, Ryu S, Kim J (2012) ULP-SRP: Ultra low power samsung reconfigurable processor for biomedical applications. In: 2012 international conference on field-programmable technology (FPT), pp 329–334. doi: 10.1109/FPT.2012.6412157
52. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680. doi: 10.1126/science.220.4598.671
53. Koch D, Beckhoff C, Teich J (2009) Minimizing internal fragmentation by fine-grained two-dimensional module placement for runtime reconfigurable systems. In: 17th annual IEEE symposium on field-programmable custom computing machines (FCCM 2009). IEEE Computer Society, pp 251–254
54. Koch D, Beckhoff C, Tørrison J (2010) Advanced partial run-time reconfiguration on spartan-6 fpgas. In: 2010 international conference on field-programmable technology (FPT), pp 361–364
55. Koch D, Beckhoff C, Wold A, Torresen J (2013) Easypr – an easy usable open-source PR system. In: 2013 international conference on field-programmable technology (FPT), pp 414–417
56. Kurdahi F, Parker A (1987) Real: a program for register allocation. In: 24th conference on design automation, pp 210–215. doi: 10.1109/DAC.1987.203245
57. Langhammer M, Pasca B (2015) Floating-point DSP block architecture for FPGAs. In: Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays, FPGA '15. ACM, New York, pp 117–125. doi: 10.1145/2684746.2689071
58. Lanuzza M, Perri S, Corsonello P, Margala M (2007) A new reconfigurable coarse-grain architecture for multimedia applications. In: 2007 second NASA/ESA conference on adaptive hardware and systems, AHS 2007, pp 119–126
59. Lattner C (2002) LLVM: an infrastructure for multi-stage optimization. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana. http://llvm.org/pubs/2002-12-LattnerMSThesis.pdf
60. Lee Je, Choi K, Dutt ND (2003) An algorithm for mapping loops onto coarse-grained reconfigurable architectures. SIGPLAN Not 38(7):183–188
61. Lee G, Lee S, Choi K (2008) Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques. In: Proceedings of the international SoC design conference, ISOCC '08, vol 01, pp I–395–I–398
62. Lee D, Jo M, Han K, Choi K (2009) Flora: coarse-grained reconfigurable architecture with floating-point operation capability. In: 2009 international conference on field-programmable technology, FPT 2009, pp 376–379
63. Lee G, Choi K, Dutt N (2011) Mapping multi-domain applications onto coarse-grained reconfigurable architectures. IEEE Trans Comput-Aided Des Integr Circuits Syst 30(5):637–650
64. Lee H, Nguyen D, Lee J (2015) Optimizing stream program performance on cgra-based systems. In: Proceedings of the 52nd annual design automation conference, DAC '15. ACM, New York, pp 110:1–110:6
65. Leijten J, Burns G, Huisken J, Waterlander E, van Wel A (2003) AVISPA: a massively parallel reconfigurable accelerator. In: Proceedings of the 2003 international symposium on system-on-chip, pp 165–168
66. Li S, Ebeling C (2004) Quickroute: a fast routing algorithm for pipelined architectures. In: Proceedings of the 2004 IEEE international conference on field-programmable technology, pp 73–80. doi: 10.1109/FPT.2004.1393253
67. Liang C, Huang X (2008) Smartcell: a power-efficient reconfigurable architecture for data streaming applications. In: 2008 IEEE workshop on signal processing systems, SiPS 2008, pp 257–262

68. Lie W, Feng-yan W: Dynamic partial reconfiguration in FPGAs. In: 2009 third international symposium on intelligent information technology application, IITA 2009, vol 2, pp 445–448 (2009)
69. Lysaght P, Blodget B, Mason J, Young J, Bridgford B (2006) Invited paper: enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of Xilinx FPGAs. In: FPL, pp 1–6. IEEE
70. Marshall A, Stansfield T, Kostarnov I, Vuillemin J, Hutchings B (1999) A reconfigurable arithmetic array for multimedia applications. In: Proceedings of the 1999 ACM/SIGDA seventh international symposium on field programmable gate arrays, FPGA '99. ACM, New York, pp 135–143
71. McMurchie L, Ebeling C (1995) Pathfinder: a negotiation-based performance-driven router for FPGAs. In: Proceedings of the third international ACM symposium on field-programmable gate arrays, FPGA '95, pp 111–117. doi: 10.1109/FPGA.1995.242049
72. Mehta N (2015) Ultrascale architecture: highest device utilization, performance, and scalability, WP455 (v1.2), October 29, 2015
73. Mei B, Vernalde S, Verkest D, De Man H, Lauwereins R (2003) Adres: an architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In: Cheung PYK, Constantinides G (eds) Field programmable logic and application. Lecture notes in computer science, vol 2778. Springer, Berlin/Heidelberg, pp 61–70
74. Mei B, Vernalde S, Verkest D, De Man H, Lauwereins R (2003) Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In: Proceedings of the conference on design, automation and test in Europe – DATE '03, vol 1. IEEE Computer Society, Washington, DC, p 10296
75. Mei B, Vernalde S, Verkest D, Lauwereins R (2004) Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: a case study. In: Proceedings of the 2004 design, automation and test in Europe conference and exhibition, vol 2, pp 1224–1229
76. Mei B, Lambrechts A, Mignolet JY, Verkest D, Lauwereins R (2005) Architecture exploration for a reconfigurable architecture template. IEEE Des Test Comput 22(2):90–101
77. MicroBlaze processor: MicroBlaze soft processor core (2012). http://www.xilinx.com/products/design-tools/microblaze.html
78. Mirsky E, DeHon A (1996) Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In: Proceedings of the IEEE symposium on FPGAs for custom computing machines, pp 157–166
79. Miyamori T, Olukotun K (1998) Remarc: reconfigurable multimedia array coprocessor. In: IEICE transactions on information and systems E82-D, pp 389–397
80. Moghaddam MS, Paul K, Balakrishnan M (2013) Design and implementation of high performance architectures with partially reconfigurable cgras. In: 2013 IEEE 27th international parallel and distributed processing symposium workshops PhD forum (IPDPSW), pp 202–211. doi: 10.1109/IPDPSW.2013.121
81. Moghaddam MS, Paul K, Balakrishnan M (2014) Mapping tasks to a dynamically reconfigurable coarse grained array. In: 2014 IEEE 22nd annual international symposium on field-programmable custom computing machines (FCCM), pp 33–33. doi: 10.1109/FCCM.2014.20
82. Moghaddam M, Balakrishnan M, Paul K (2015) Partial reconfiguration for dynamic mapping of task graphs onto 2d mesh platform. In: Sano K, Soudris D, Hübner M, Diniz PC (eds) Applied reconfigurable computing. Lecture notes in computer science, vol 9040. Springer, pp 373–382
83. Moon J, Moser L (1965) On cliques in graphs. Isr J Math 3(1):23–28. doi: 10.1007/BF02760024
84. MultiTrack interconnect in Stratix III devices (2009)
85. Oh T, Egger B, Park H, Mahlke S (2009) Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. SIGPLAN Not 44(7):21–30
86. Park S, Choi K (2011) An approach to code compression for CGRA. In: 2011 3rd Asia symposium on quality electronic design (ASQED), pp 240–245. doi: 10.1109/ASQED.2011.6111753

87. Park IC, Kyung CM (1991) Fast and near optimal scheduling in automatic data path synthesis. In: 28th ACM/IEEE design automation conference, pp 680–685
88. Park H, Fan K, Kudlur M, Mahlke S (2006) Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In: Proceedings of the 2006 international conference on compilers, architecture and synthesis for embedded systems, CASES '06. ACM, New York, pp 136–146
89. Park H, Fan K, Mahlke SA, Oh T, Kim H, Kim HS (2008) Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques, PACT '08. ACM, New York, pp 166–176
90. Park JJK, Park Y, Mahlke S (2013) Efficient execution of augmented reality applications on mobile programmable accelerators. In: 2013 international conference on field-programmable technology (FPT), pp 176–183. doi: 10.1109/FPT.2013.6718350
91. Patel K, Bleakley CJ (2010) Systolic algorithm mapping for coarse grained reconfigurable array architectures. In: Proceedings of the 6th international conference on reconfigurable computing: architectures, tools and applications, ARC'10, pp 351–357. Springer, Berlin/Heidelberg
92. Patel K, McGettrick S, Bleakley CJ (2011) Syscore: a coarse grained reconfigurable array architecture for low energy biosignal processing. In: 2011 IEEE 19th annual international symposium on field-programmable custom computing machines (FCCM), pp 109–112
93. Paul K, Dash C, Moghaddam M (2012) reMORPH: a runtime reconfigurable architecture. In: 2012 15th Euromicro conference on digital system design (DSD), pp 26–33
94. Pinson E, Prins C, Rullier F (1994) Using tabu search for solving the resource-constrained project scheduling problem. In: Proceedings of the 4th international workshop on project management and scheduling, Leuven, pp 102–106
95. Rau BR (1994) Iterative modulo scheduling: an algorithm for software pipelining loops. In: Proceedings of the 27th annual international symposium on microarchitecture, MICRO 27. ACM, New York, pp 63–74. doi: 10.1145/192724.192731
96. Salefski B, Caglar L (2001) Re-configurable computing in wireless. In: Proceedings of 2001 design automation conference, pp 178–183
97. Sanchez E, Sterpone L, Ullah A (2014) Effective emulation of permanent faults in asics through dynamically reconfigurable FPGAs. In: 2014 24th international conference on field programmable logic and applications (FPL), pp 1–6
98. Sato T, Watanabe H, Shiba K (2005) Implementation of dynamically reconfigurable processor dapdna-2. In: 2005 IEEE VLSI-TSA international symposium on VLSI design, automation and test (VLSI-TSA-DAT), pp 323–324
99. Sato T, Watanabe H, Shiba K: Implementation of dynamically reconfigurable processor dapdna-2. In: 2005 IEEE VLSI-TSA international symposium on VLSI design, automation and test (VLSI-TSA-DAT), pp 323–324 (2005)
100. Singh H, Lee MH, Lu G, Kurdahi F, Bagherzadeh N, Chaves Filho E (2000) Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans Comput 49(5):465–481
101. Smit GJM, Kokkeler ABJ, Wolkotte PT, Hölzenspies PKF, van de Burgwal MD, Heysters PM (2007) The chameleon architecture for streaming DSP applications. EURASIP J Embed Syst 2007(1):1–10
102. SYSTEMS T (2016) Coarse-grained reconfigurable architecture. http://www.trentonsystems.com/blog/intel-cpu-computing/moores-law-pushing-processor-technology-to-14-nanometers/
103. Tehre V, Kshirsagar R (2012) Survey on coarse grained reconfigurable architectures. Int J Comput Appl 48(16):1–7. Full text available
104. Todman T, Constantinides G, Wilton S, Mencer O, Luk W, Cheung P (2005) Reconfigurable computing: architectures and design methods. IEE Proc Comput Digital Tech 152(2):193–207
105. Trimberger S (2015) Three ages of FPGAs: a retrospective on the first thirty years of FPGA technology. Proc IEEE 103(3):318–331

106. Tsu W, Macy K, Joshi A, Huang R, Walker N, Tung T, Rowhani O, George V, Wawrzynek J, DeHon A (1999) HSRA: high-speed, hierarchical synchronous reconfigurable array. In: Proceedings of the 1999 ACM/SIGDA seventh international symposium on field programmable gate arrays, FPGA '99. ACM, New York, pp 125–134
107. University S (2016) SUIF compiler system. http://suif.stanford.edu/
108. Villarreal J, Park A, Najjar W, Halstead R (2010) Designing modular hardware accelerators in C with ROCCC 2.0. In: 2010 18th IEEE annual international symposium on field-programmable custom computing machines (FCCM), pp 127–134. doi: 10.1109/FCCM.2010.28
109. Vivado HLS: Xilinx Vivado Design Suite, Inc. (2012). http://www.xilinx.com/products/design-tools/vivado.html
110. Waingold E, Taylor M, Srikrishna D, Sarkar V, Lee W, Lee V, Kim J, Frank M, Finch P, Barua R, Babb J, Amarasinghe S, Agarwal A (1997) Baring it all to software: raw machines. Computer 30(9):86–93
111. Watkins MA, Albonesi DH (2010) ReMAP: a reconfigurable heterogeneous multicore architecture. In: Proceedings of the 2010 43rd annual IEEE/ACM international symposium on microarchitecture, MICRO '43. IEEE Computer Society, Washington, DC, pp 497–508
112. Xcell Journal Issue 52:. http://www.xilinx.com/publications/archives/xcell/Xcell52.pdf
113. Xilinx 7 series FPGA. www.xilinx.com. Accessed 27 Nov 2015
114. Xilinx (2012) Virtex 5 FPGA user guide
115. Xilinx (2014) 7 series FPGAs configurable logic block
116. Yoon JW, Shrivastava A, Park S, Ahn M, Jeyapaul R, Paek Y (2008) SPKM: a novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In: 2008 Asia and South Pacific design automation conference, pp 776–782. doi: 10.1109/ASPDAC.2008.4484056
117. Yoon J, Shrivastava A, Park S, Ahn M, Paek Y (2009) A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. IEEE Trans Very Large Scale Integr (VLSI) Syst 17(11):1565–1578

Application-Specific Processors

12

Tulika Mitra

Abstract

General-Purpose Processors (GPPs) and Application-Specific Integrated Circuits (ASICs) are the two extreme choices for computational engines. GPPs offer complete flexibility but are inefficient both in terms of performance and energy. In contrast, ASICs are highly energy-efficient, provide the best performance at the cost of zero flexibility. Application-specific processors or custom processors bridge the gap between these two alternatives by bringing in improved power-performance efficiency within the familiar software programming environment. An application-specific processor architecture augments the base instruction-set architecture with customized instructions that encapsulate the frequently occurring computational patterns within an application. These custom instructions are implemented in hardware enabling performance acceleration and energy benefits. The challenge lies in inventing automated tools that can design an application-specific processor by identifying and implementing custom instructions from the application software specified in high-level programming languages. In this chapter, we present the benefits of application-specific processors, their architecture, automated design flow, and the renewed interests in this class of architectures from energy-efficiency perspective.

Acronyms

ALU	Arithmetic-Logic Unit
ASIC	Application-Specific Integrated Circuit
BERET	Bundled Execution of REcurring Traces
CAD	Computer-Aided Design
CCA	Configurable Compute Accelerator
CFG	Control-Flow Graph

T. Mitra (✉)
Department of Computer Science, School of Computing, National University of Singapore, Singapore, Singapore
e-mail: tulika@comp.nus.edu.sg

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_13

CFU	Custom Functional Unit
CIS	Custom Instruction-Set
DFG	Data-Flow Graph
DISC	Dynamic Instruction-Set Computer
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
ILP	Integer Linear Program
IR	Intermediate Representation
ISA	Instruction-Set Architecture
ISEF	Stretch Instruction-Set Extension Fabric
MAC	Multiply-Accumulator
MIMO	Multiple Input Multiple Output
MISO	Multiple Input Single Output
PFU	Programmable Functional Unit
PRISC	Programmable Instruction-Set Processor
RAM	Random-Access Memory
RISC	Reduced Instruction-Set Processor
RISPP	Rotating Instruction-Set Processing Platform
SFU	Specialized Functional Unit
VLIW	Very Long Instruction Word

Contents

12.1 Introduction

Over the years, the General-Purpose Processors (GPPs) has been established as the de facto choice for execution engine. Microprocessors – single chip GPPs – were invented in 1971 and have enjoyed unprecedented performance growth aided by technology scaling (Moore's law [47] for transistor density) and microarchitectural innovations (out-of-order execution, branch prediction, speculation, cache memory) [53]. GPPs are designed to support a wide range of applications through software programmability at reasonable power-performance point. The software is the key here to provide application-specific functionality or specialization.

The generic nature of GPPs is also the reason behind their inefficiency, both in terms of power and performance [30]. The GPPs need to support a comprehensive Instruction-Set Architecture (ISA) as the abstraction and interface to the software. But processing a computation expressed in a generic ISA involves significant overhead in the front end of the processor pipeline – such as instruction fetch, instruction decode, and register access – that does not contribute toward the core computations, which are the arithmetic or logical operations.

At the other end of the spectrum, we have the Application-Specific Integrated Circuits (ASICs) as the completely specialized execution engines. ASICs can provide unprecedented power-performance benefits compared to the GPPs as they completely eliminate the instruction processing overhead and only perform the required computations. But the efficiency comes at the cost of flexibility as ASICs does not provide any programmability (hence the name non-programmable accelerators for computations implemented in an ASIC). Any change in the application incurs complete redesign and fabrication cost. Thus ASICs are only suitable for critical computational kernels, such as video encoding/decoding, whose performance per watt requirements cannot be met through the GPPs.

The domain-specific processors offer an interesting design choice between the two extreme alternative of GPPs and ASICs. Graphics Processing Units (GPUs) and Digital Signal Processors (DSPs) are well-established examples of domain-specific processors. The ISA and the microarchitecture are carefully designed to accommodate the applications in a specific domain. Thus domain-specific processors attempt to balance specialization with programmability. Still these processors cannot possibly cover all the application domains, and programming them is not as straightforward as GPPs.

In this chapter, we will focus on an interesting and compelling alternative called the *application-specific processors* or *custom processors* [34]. We will use the terms application-specific processors and custom processors interchangeably in this chapter. An application-specific processor is a general-purpose programmable processor that has been configured and extended with respect to a particular application or application domain [24]. Configurability refers to the ability to select and set the size of different microarchitectural features such as number and types

of functional units, register file size, instruction and data cache size, associativity, etc. according to the characteristics of the application domain. For example, if an application does not use floating-point operations, the floating-point functional units can be eliminated from the underlying microarchitecture. Optimal setting of the configuration parameters for an application domain is a complex design space exploration problem [38] that has been investigated thoroughly [20, 29, 51] in the past decade.

However, in this chapter, we will focus on the extensibility part of the application-specific processors where the instruction-set architecture of the general-purpose processor is augmented with application-specific *Instruction-set extensions*, also known as the *custom instructions* . The custom instructions capture the key computation fragments or patterns within an application. The custom instructions are added to the base ISA of the general-purpose processor. The computation corresponding to each custom instruction is synthesized as a *Custom Functional Unit (CFU)*, and all the CFUs are included in the processor's data path alongside existing functional units, such as Arithmetic-Logic Units (ALUs), multipliers, and so on. The front end of the processor pipeline, for example, the decode stage, needs to be suitably changed to take in these new instructions. Similarly, the compiler and associated software tool chain are modified to support the custom instructions such that a new application can exploit additional instructions in the ISA. As a custom instruction combines a number of base instructions, it amortizes the front-end processing overhead. Moreover, the synthesis process of the CFU can be optimized so that the operations within a CFU can be parallelized and chained together to offer a very competitive critical path delay that is far shorter than the sum of the delays of the individual operations. These factors together lead to substantially improved performance and energy efficiency for application-specific processors compared to the GPPs. Thus, application-specific processors offer an easy and incremental path toward specialization, while still staying within the comfortable and familiar software programming environment of the GPPs.

Initially, application-specific processors were enthusiastically and successfully adopted in the embedded systems domain. They are an excellent match for this domain because an embedded system is designed to provide a well-defined set of functionalities throughout its lifetime. Thus the custom instructions can be constructed to accelerate the specific computations involved in providing the required functionality. The focus at that time has been on the automated design of the custom instructions. More concretely, given an application, how do we automatically enumerate the potential custom instructions (*custom instructions enumeration*) and choose the appropriate ones under area, energy, and/ or performance constraints (*custom instructions selection*)? A flurry of research activity in the past 15 years on design automation of custom processors has made significant advances, even though some open problems and challenges still remain unresolved. A number of processor vendors have offered commercial customizable processor along with the complete automation tool chain to enumerate, select, and synthesize the custom instructions, followed by the synthesis of the application-specific processor including the custom instructions, and the compiler to fully realize the potential of the custom instructions

from software with minimal engagement from the programmers. The Xtensa customizable processors from Tensilica [39] are the best examples of this design paradigm. In general, however, the interest in application-specific processors has been primarily restricted to the embedded systems area till very recently.

In the past five years or so, a number of technological challenges have brought the application-specific processors to the forefront even in the general-purpose and high-performance computing domains. First, the energy efficiency rather than performance has increasingly become the first-class design goal [48] for any system, be it battery-powered embedded and mobile platforms, or high-performance servers with continuous access to electrical power sockets. Second, the failure of Dennard scaling [19] back in around 2005 has had a devastating effect on the microprocessor design. As power per transistor does not scale any more with feature scaling from one technology generation to another, increasing transistor density following Moore's law leads to increasing power density for the chip. Thus the core has to operate at a frequency that is lower than the default frequency enabled through technology scaling, so as to keep the power density of the chip within acceptable limits. Moreover, complex microarchitectural features have long ceased to provide any further performance improvements [50] as we have hit the instruction-level parallelism wall [67] and the memory wall [69]. This loss of frequency scaling and microarchitectural enhancement have kept the single-core performance at a standstill for the past ten years or so. Instead, the abundance of on-chip transistors as per Moore's law has been employed to build multi- or even many-core chips consisting of identical general-purpose processor cores [22]. The multi-cores are perfect match for applications with high thread-level parallelism. But the sequential fragment of the application still suffers from poor single-core performance and thereby limits the performance potential of the entire application according to Amdahl's law [3].

More importantly, though, multi-core scaling is also coming to an end in the near future [21]. As we increase the number of core on chip, the failure of Dennard scaling implies that the total chip power increases. But the packaging and cooling solutions are not sufficient to handle this increasing chip power. Thus we can have more core on chip; but we can only power on a subset of these cores to meet the chip power budget. This phenomenon has been termed *dark silicon* [43], where a significant fraction of the chip remains unpowered or dark. The dark silicon era naturally leads to the design of heterogeneous multi-core architectures [44] rather than the homogeneous multi-core designs prevalent today. Depending on the applications executing on the architecture, only the cores that are well suited for the current workload can be switched on at any point in time, leading to performance- and energy-efficient computing [61]. In other words, the cheap silicon area (that would have remained unused anyway due to limited power budget) is being traded to add execution engines that will be used sparingly and judiciously. Thus, the dark silicon era has automatically paved the way for specialized cores and have generated renewed interest in application-specific or custom processors [11, 62].

The objectives and challenges in designing application-specific processors are somewhat different, though, for embedded computing systems and general-purpose

high-performance computing systems. First, unlike embedded systems that execute only a fixed set of applications throughout its lifetime, general-purpose computing systems encounter a diverse set of workloads that may be unknown at design time. So, while we can profile the applications and design the best set of custom instructions to accelerate the embedded workload, the same approach is not feasible in general-purpose systems. This leads to the design of custom instructions that are somewhat parameterized and can be reused across workloads [65]. Another possible direction is the design of custom functional units or a flexible fabric that can be reconfigured to support a varied set of custom instructions [32]. Second, custom instruction enumeration in the context of embedded systems has been generally restricted to computations and has mostly excluded storage elements inside a custom instruction. The implication of this choice is that custom instructions are small and the gain in performance comes from repeated occurrence – either statically in the program code or dynamically due to its presence in a loop body – of the same custom instructions. But the performance gain and energy improvement are still modest with small custom instructions compared to the potential when large custom instructions are formed combining computations with storage elements and without any restrictions in terms of input/output operands [30, 71]. Thus bigger custom instructions with storage elements are recommended for general-purpose high-performance computing systems.

The rest of the chapter is organized as follows. We will start off with a brief overview of the architecture of the application-specific processors and the corresponding design automation flow. This will be followed by detailed discussion on custom instructions enumeration and selection algorithms. We will next present customizable architectures with dynamic or reconfigurable custom instructions and the necessary algorithms to identify custom instructions and exploit such architectures. Finally, we will provide a quick review of recent attempts to bring customization to general-purpose computing platforms.

12.2 Architectural Overview and Design Flow

In this section, we provide an overview of the application-specific processor architecture and the corresponding design flow.

12.2.1 Application-Specific Processor Architecture

The generic architecture of an application-specific processor or custom processor is shown in Fig. 12.1. The instruction-set architecture of the base processor is modified to include the application-specific instructions or custom instructions. A custom instruction encapsulates a frequently occurring computational pattern involving a number of basic operations (see Fig. 12.1b). Each custom instruction is implemented as a CFU. The pipeline data path of the base processor core is augmented with the CFUs alongside existing functional units (ALU, multiplier,

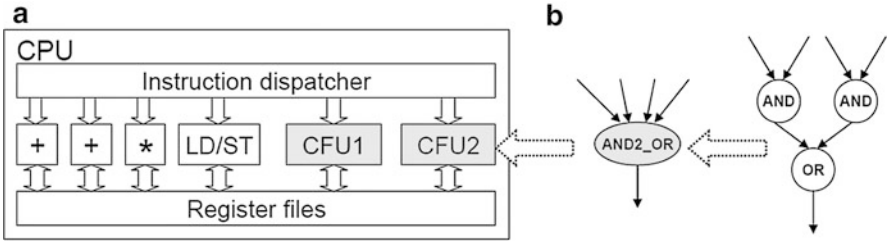


Fig. 12.1 Architecture of an application-specific processor with two CFUs

load/store units, etc.) and are treated the same way as shown in Fig. 12.1a. The CFUs can access the input and output operands stored in the register file just like regular functional units. Thus a custom instruction can be fetched, decoded, and dispatched to the respective CFU just like normal instructions. The biggest advantage of custom instructions is the improved latency and decreased power consumption to execute the computational pattern. For example, in Fig. 12.1b, the custom instruction consists of three basic operations (two AND and one OR). In the base processor core, this computation pattern requires fetch, decode, and execution of three instructions. This is reduced to only one custom instruction immediately saving the fetch, decode, dispatch time, and energy. More importantly, as the custom instruction is synthesized in hardware, the implementation can take advantage of parallelism. For example, the two AND operations can be executed in parallel. Also the critical path now consists of AND-OR operation. If the critical path can fit inside the cycle time of the base processor, the CFU can execute the custom instruction in one clock cycle. This is the case when the clock cycle time of the processor is determined by a complex operation such as Multiply-Accumulator (MAC) or even addition operation whereas the basic operations on the critical path are much simpler (such as logical operations) such that multiple of them can be chained together in a single cycle. If the critical path exceeds the cycle time, then the CFU will execute the custom instruction in multiple cycles (possible pipelined), say N, where $N = (critical\ path\ latency)/(clock\ period)$. Still, N is likely much less than M, the minimum number of cycles required to execute the basic operations in the computational pattern individually. A positive side effect of the custom instructions approach is the increased code density and hence reduced code size, which is an important issue in embedded systems. A custom instruction also reduces the number of register accesses significantly as the intermediate results (the results of the AND operations) need not be written back and read from the register file. In the example pattern in Fig. 12.1b, the custom instruction needs four register reads and one register write as opposed to six register reads and three register writes for the original three instruction sequence.

A custom instruction may require more input and output operands compared to the typical two-input, one-output basic Reduced Instruction-Set Processor (RISC) instructions. Yu and Mitra [71] showed experimentally that the performance potential of custom instructions improves with increasing number of input and

output operands. Later, Verma et al. [66] proved that under fairly weak assumptions, increasing the number of nodes in a pattern (which typically results in increased input/output operands) can never lead to reduced speedup. But supporting more input and output operands within the framework of normal processor pipeline requires some additional support from the underlying microarchitecture and the ISA, which will be discussed in Sect. 12.3.4. The maximum number of input and output operands supported per custom instruction in an architecture defines the custom instruction identification algorithm for that architecture.

12.2.2 Design Flow

The main design effort in tailoring an extensible processor is to define the custom instructions for a given application to meet certain design goals. Identifying suitable custom instructions is essentially a hardware-software partitioning problem that divides the computations between the software execution (using base instructions) and hardware execution (using custom instructions). Various design constraints must be satisfied in order to deliver a viable system, including performance, silicon area cost, power consumption, and architectural limitations. This problem is frequently modeled as a single-objective optimization where a certain aspect is optimized (e.g., performance), while the other aspects are considered as constraints.

The generic design flow for application-specific processors is presented in Fig.12.2. The input to this design flow is the reference software implementation of the application written in a high-level programming language such as C or C++. In the application-specific processor design flow, the compiler performs additional steps toward customizing the base processor core: identifying the computational patterns that can potentially serve as custom instructions, selecting a subset of these patterns under various constraints to maximize the objective function, and finally synthesizing the new CFUs and generating the binary executables under the new instruction set. This automated hardware-software codesign approach ensures that large programs can be explored and the software programmers can easily adapt to the design flow without in-depth hardware knowledge.

In a generic compiler, high-level language statements are first transformed by the compiler front end to an Intermediate Representation (IR). Various machine-independent optimizations are carried out on the IR. Then, the back end of the compiler generates binary executables for the target processor by binding IR objects to actual architectural resources: operations to instructions, operands to registers or memory locations, and concurrencies and dependencies to time slots, through instruction binding, register allocation, and instruction scheduling, respectively. Various machine-dependent optimizations are also performed in the back end. The custom instruction identification, selection, and binary executable generation with custom instructions are all performed in the back end on the IR.

The IR consists of a Control-Flow Graph (CFG) and a Data-Flow Graph (DFG) (also called a data-dependency graph). The CFG is a graph structure where the nodes are the basic blocks – sequence of instructions with a single-entry and single-exit

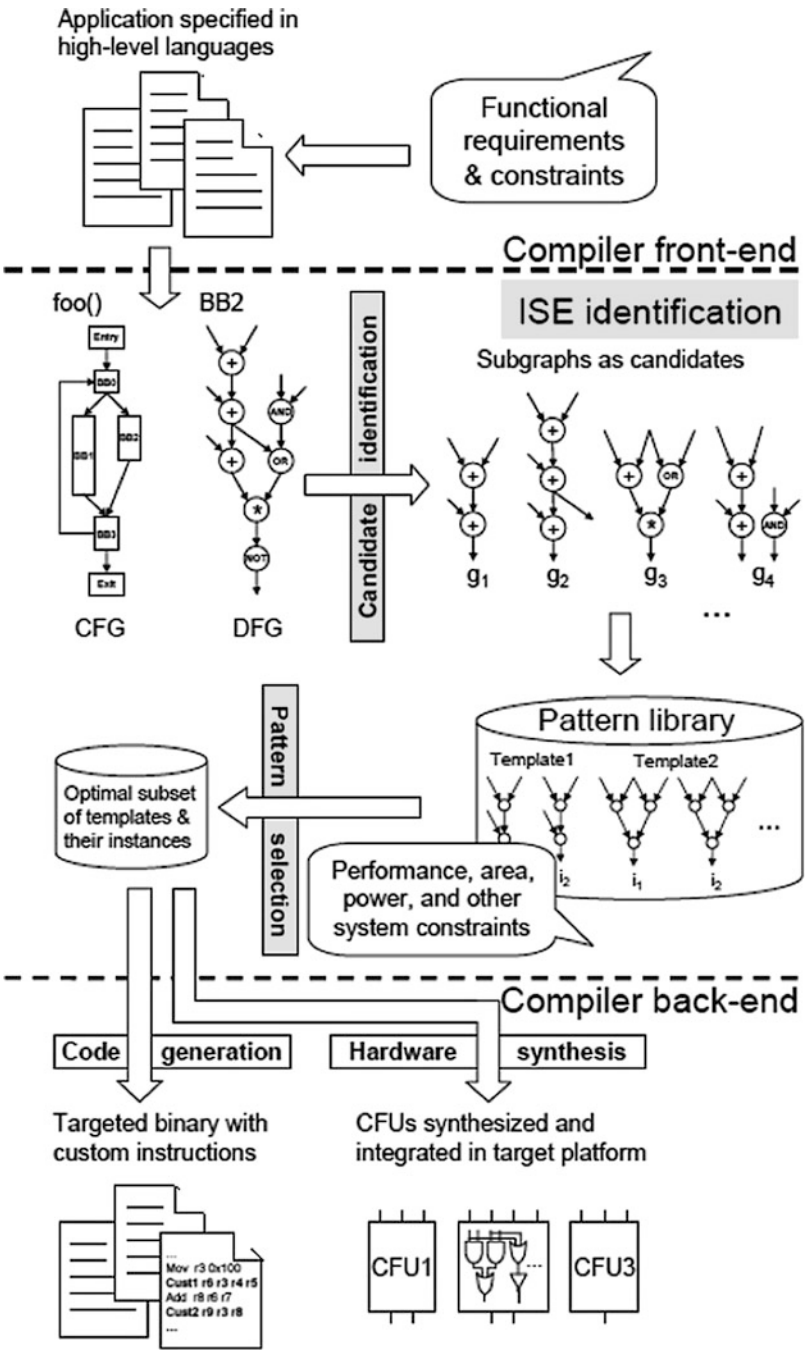


Fig. 12.2 The design flow for application-specific processors [52]

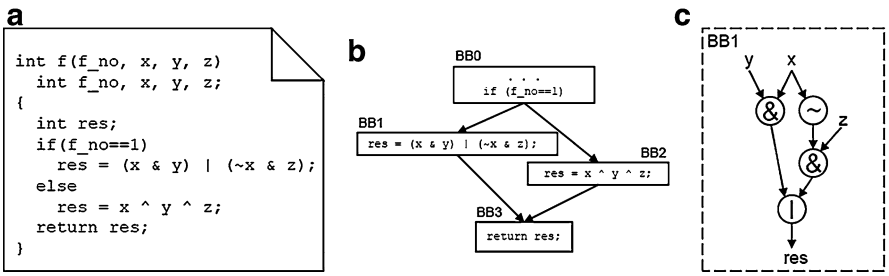


Fig. 12.3 Control-flow graph and data-flow graph

point. The edges among the basic block represent control-flow transfer from one basic block to another through conditional statements, loops, function calls, etc. For each basic block, the computation within a basic block is captured by a DFG where the nodes represent the operations and the edges represent dependency among the operations. Figure 12.3 shows an example of CFG and DFG corresponding to a code segment. In the base processor, each operation in the DFG typically corresponds to an instruction in the ISA. However, a custom instruction can cover a cluster of operations and is hence captured as a subgraph of the DFG.

Normally, custom instructions identification is restricted to the DFG within each basic block. However, basic blocks are usually quite small consisting of only few instructions. Thus there is limited opportunity to extract large custom instructions within basic blocks and obtain significant performance improvement. A limit study by Yu and Mitra [71] showed significant benefit in identifying custom instructions across basic block boundaries. Thus it is essential to create larger blocks containing multiple basic blocks, for example, traces, superblocks, and hyperblocks, and provide architectural support to for these extended blocks. In that case, the DFG can be built for these larger blocks.

The custom instruction identification is essentially a subgraph enumeration problem. For each DFG, the computational patterns that satisfy certain constraints are enumerated as potential custom instruction candidates. If a pattern appears multiple times either within the same basic block or different basic blocks, these are considered as different instances of the same computation pattern. This step builds a library of potential candidate patterns. The next step evaluates the different patterns and selects a subset that optimizes certain goals under the different constraints. This step requires profiling data. The application is profiled on the base processor with representative input data sets. The profiling identifies the *hot spots* where most of the computation time is spent, and these hot spots are ideal candidates for implementation as custom instructions and may benefit from faster execution on CFUs. The performance benefit of each pattern combined with the frequency of execution of that pattern from profiling information is used in the custom instructions selection process.

Finally, the selected patterns are passed on to the last step of the design framework. Each selected pattern is synthesized in hardware as a CFU, and the CFUs are added to the data path of the processor. The processor control is modified to accommodate these new instructions. The new instructions are acknowledged in the instruction binding stage of the compiler either by simple peephole substitution or by the pattern matcher to produce an executable that exploits the custom instructions.

12.3 Custom Instructions Identification and Selection

We start off this section by first presenting the terminology and definitions related to custom instructions identification from a data-flow graph.

12.3.1 Formal Definitions

The custom instructions are identified on the data-flow graphs corresponding to the basic blocks of a program. A *DFG* G is a directed acyclic graph that captures the flow of data within a basic block. The set of nodes or vertices $V(G)$ represent the operations, while the set of edges $E(G)$ represent the flow of data among the operations. An edge $e = (u, v)$ where $e \in E(G)$, $u, v \in V(G)$ denotes a data dependency from u to v where v can execute only after u completes execution. Figure 12.4 shows a data-flow graph consisting of the blue and the red nodes.

Each graph G is also associated with a supergraph G^+ that contains additional nodes V^+ and edges E^+. The additional nodes V^+ represent the input and output variables of the basic block, while the additional edges E^+ connect the nodes in V^+ to the nodes in $V(G)$. The green nodes in Fig. 12.4 correspond to the additional

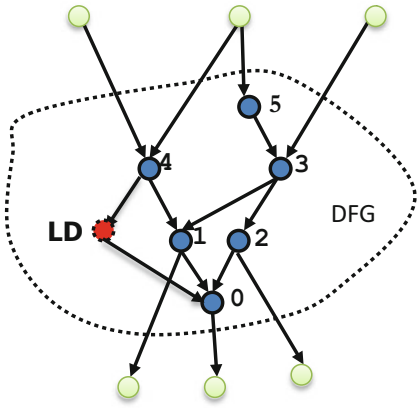


Fig. 12.4 Data-flow graph

nodes to create the supergraph corresponding to the DFG. The DFG requires three input variables and three output variables.

Given an edge $e = (u, v)$, the node u is called the *immediate predecessor* of node v, while v is the *immediate successor* of u. Let us define $IPred(v) = \{u|(u, v) \in E(G)\}$ and $ISucc(v) = \{u|(v, u) \in E(G)\}$ as the *immediate predecessor set* and *immediate successor set* of node v, respectively. The in-degree and out-degree of node v are $|IPred(v)|$ and $|ISucc(v)|$, respectively. A *path* $v_0 \rightsquigarrow v_n$ is a sequence $\langle v_0, v_1, \ldots, v_n \rangle$ of nodes where $v_i \in V(G)$ for $0 \leq i \leq n$ such that $(v_i, v_{i+1}) \in E(G)$ for $0 \leq i \leq (n-1)$. We define predecessor set $Pred(v)$ as the set of nodes that can reach v through a path in the graph, i.e., $Pred(v) = \{u|u \rightsquigarrow v \in G\}$. Similarly, we define successor set $Succ(v)$ as the set of nodes that can be reached from v through a path in the graph, i.e., $Succ(v) = \{u|v \rightsquigarrow u \in G\}$.

A *source* node in the supergraph G^+ has no predecessor (zero in-degree), while a *sink* node has no successor (zero out-degree). The source nodes represent the input operands (including immediate operands), while the sink nodes represent the output operands corresponding to the DFG. Together the source nodes and the sink nodes correspond to V^+. The remaining nodes $V(G)$ are the *internal* nodes of the data-flow graph that represent the operations (arithmetic and logical operations, load/store, etc.) supported by the ISA of the baseline processor. The green nodes in Fig. 12.4 are the source and sink nodes, while the blue and red nodes are the internal nodes of the data-flow graph.

A custom instruction or a *pattern* C is a subgraph of the DFG G induced by the set of vertices $C \subseteq V(G)$. The subgraph consists of vertices $V(C) \subseteq V(G)$ and edges $\{(u, v) \in E(G)|u, v \in V(C)\}$. For example, the set of vertices $\{0, 1, 2\}$ form a pattern. A node u is an *input* of pattern C if $u \notin V(C)$, $v \in V(C)$, and there exists an edge $(u, v) \in E(G^+)$. Similarly, a node u is an *output* of pattern C if $u \in V(C)$, $v \notin V(C)$, and there exists an edge $(u, v) \in E(G^+)$. Let $In(C)$ and $Out(C)$ be the set of input and output nodes of pattern C, respectively. The input nodes represent the input values or variables used by the custom instruction, while the output nodes present values produced by the custom instruction that will be used by other operations, either in G or in another basic block. Many architectures impose constraints on the number of inputs and outputs per custom instruction, known as the *I/O constraint*. The nodes $3, 4, LD$ are the input and nodes $0, 1, 2$ are all output corresponding to the pattern $\{0, 1, 2\}$ in Fig. 12.4.

An architecture may impose restrictions on the kind of operations that may be included as part of a custom instruction. Most architectures do not allow memory operations (load/store) and control operations (branch) to be part of custom instructions. A node is *valid* if it can be part of a custom instruction; otherwise it is *invalid*. By definition, the source and the sink nodes are not part of custom instructions and hence are invalid. Internal nodes can be invalid too if the corresponding operation cannot be accommodated within a custom instruction. For example, the red node in Fig. 12.4 corresponds to a load operation, and it is an invalid node in addition to the green source/sink nodes. Let $X(G) \in V(G)$ be the set of internal invalid nodes of the DFG.

The following special patterns are of interest in custom instructions enumeration problem.

Connected pattern A pattern C is connected if for any pair of nodes $u, v \in V(C)$ in the pattern, there exists a path between u and v within the pattern in the undirected graph that underlies the directed induced subgraph of C. $\{0, 1, 2\}$ is a connected pattern in Fig. 12.4.

Disjoint pattern A pattern is disjoint if it is not connected. A disjoint pattern consists of two or more connected patterns. The pattern $\{3, 4, 5\}$ is a disjoint pattern consisting of two connected patterns $\{4\}$ and $\{3, 5\}$.

MISO pattern A pattern C with only one output ($|Out(C) = 1|$) is called a Multiple Input Single Output (MISO) pattern. Clearly, a MISO pattern should be a connected pattern. $\{3, 5\}$ is a MISO pattern with 3 as the output. Note that $\{0, 1, 2\}$ is not a MISO pattern as it has three outputs.

MIMO pattern A pattern with multiple input and multiple output is called a Multiple Input Multiple Output (MIMO) pattern. We can further distinguish between connected MIMO pattern and disjoint MIMO pattern.

Convex pattern A pattern C is convex if any intermediate node t on any path in the DFG G from a node $u \in V(C)$ to a node $v \in V(C)$ belongs to C, i.e., $t \in V(C)$. A pattern is non-convex if there exist at least one path in the DFG G from a node $u \in V(C)$ to a node $v \in V(C)$ that contains an intermediate node $t \in V(G) \setminus V(C)$. A non-convex pattern is infeasible because it cannot be executed as a custom instruction in an atomic fashion. The pattern $\{0, 1, 2, 4\}$ is a non-convex pattern because there is a path from 4 to 0 that contains the invalid LD node.

Valid pattern A pattern C is a valid custom instruction candidate if (a) the pattern does not contain any invalid node $V(C) \cap X(G) = \phi$, (b) the pattern is convex, and (c) the pattern satisfies the I/O constraints imposed by the architecture, i.e., $In(C) \leq N_{in}$ and $Out(C) \leq N_{out}$ where N_{in} and N_{out} are the maximum number of inputs and outputs allowed per custom instruction, respectively.

Maximal valid pattern A pattern C is a maximal valid pattern if it is a valid pattern, and there is no $v \in V(G) \setminus V(C)$ such that the subgraph induced by the set of vertices $(V(C) \cup v)$ is a valid pattern. For example, $\{0, 1, 2, 3, 5\}$ is a maximal valid pattern.

We will primarily concentrate on techniques to enumerate connected patterns as the disjoint patterns can be easily constructed from the connected patterns.

In the context of pattern enumeration, it is useful to define *topologically sorted level* of the nodes in the directed acyclic graph G. Each node in $V(G)$ that is only connected to the source nodes V^+ has level 0. The level of any other node $level(v) = l$ if the longest path (in terms of number of edges) from some source node to v is of length $l + 1$. We can also define the order of the nodes in G according to the topological sort; if G contains an edge $e = (u, v)$, then v should appear after

u in this *topologically sorted order*. In the example DFG of Fig. 12.4, nodes 4, 5 belong to level 0, node 3 belongs to level 1, and nodes 1, 2 belong to level 2, and node 0 belongs to level 3. A topologically sorted order for the valid nodes of the DFG is 5, 4, 3, 2, 1, 0.

12.3.2 Enumeration of MISO Patterns

The simplest custom instruction enumeration problem is the one of identifying maximal MISO (MaxMISO) patterns. A linear-time algorithm to identify MaxMISO patterns has been presented in [2]. Note than a MaxMISO pattern has a single output. So it is efficient to start with the sink nodes and proceed level by level to the source nodes. We initialize a MaxMISO pattern $C = \{v\}$ with any node v that has only one output. We can then iteratively add in the predecessors' nodes $IPred(v)$ at the next level to the pattern C as long as $u \in IPred(v)$ does not contribute a new output to the pattern or u is an invalid node ($u \in X$). The process can continue with the predecessors of the newly added nodes in the pattern till we have no more predecessors to consider. As the intersection of MaxMISOs should be empty, i.e., two MaxMISOs cannot overlap, it is easy to see that the algorithm will have linear time. In contrast, Cong et al. [18] identify all valid MISO patterns. This problem, in the most general case, has exponential complexity because each node can potentially be included or excluded from a candidate pattern. Thus, Cong et al. [18] impose restrictions on number of input operands and/or area constraint to limit the number of candidate patterns resulting in efficient pattern generation.

12.3.3 Exhaustive Enumeration of All Valid Patterns

The exhaustive enumeration of all possible valid connected patterns of a DFG G under the convexity and a specified I/O constraint is quite challenging. At first glance, in the worst case, the number of possible patterns can be $(2^{|V(G)|}$ as each vertex can be either included or excluded to form a pattern. But closer examination of the problem reveals that most of the patterns are not valid due to convexity and/or I/O constraints. Chen at al. [13] and Bonzini and Pozzi [9] have proven that the number of such valid patterns for a graph G with bounded in-degree (which is true for data-flow graphs of programs) is at most $|V(G)|^{N_{in}+N_{out}}$. If the I/O constraint is quite tight, then the enumeration is fast. A number of algorithms have been proposed in the literature to solve this problem efficiently. Gutin et al. [28] have designed an algorithm with worst-case complexity of $O(|E(G)| \times N_{in}^2 \times (n + |V(G)|^{N_{out}}))$ where n is the number of valid patterns. This bound is theoretically optimal if the number of valid patterns n is asymptotically smaller than $|V(G)|^{N_{in}+N_{out}}$. As a follow-up work, Reddington et al. [59] have proposed a version of this algorithm that has $|V(G)|^{N_{in}+N_{out}+1}$ worst-case complexity, but runs much faster in practice. At the

point of this writing, the algorithm by Reddington et al. [59] is known to be the fastest algorithm in practice for exhaustive enumeration of all valid convex patterns under I/O constraints.

In the following, we present a few representative algorithms so that the readers can better appreciate the problem and the possible solutions.

12.3.3.1 Search-Tree-Based Enumeration Algorithm

We present the first and the simplest algorithm proposed for this purpose by Atasu et al. [7,55], which is based on a search tree. It has been later shown by Reddington and Atasu [58] that the search-tree-based exhaustive algorithm has worst-case complexity of $|V(G)|^{N_{in}+N_{out}+1}$, which is quite close to the theoretical complexity.

The main insight behind the search-tree-based algorithm is that if a pattern C is not convex, then adding in the nodes that appear in lower levels in G (according to the topological sort) compared to the nodes in C would not make the resulting pattern convex. For example, the pattern $\{0, 1, 3\}$ in Fig. 12.4 is non-convex. Including the nodes 4 and 5 to this pattern will not remove the non-convexity.

Similarly, if a pattern C violates the output constraint, then adding in the nodes that appear in lower levels in G (according to the topological sort) compared to the nodes in C would not decrease the number of output operands. For example, the pattern $\{0, 1, 2\}$ in Fig. 12.4 requires three output operands. Adding the nodes that are at lower level in topological sort 3, 4, 5 to this pattern will not reduce the number of output operands of the resulting pattern. In other words, we can easily prune away all such patterns without examining them explicitly. This effective pruning of the search space leads to the efficiency of the algorithm.

The search tree is a binary tree of nodes that represent the possible patterns. The root represents an empty pattern. The nodes are added in this tree in *reverse topological order*. Let the order of the nodes of G in reverse topological order by $v_1, v_2, \ldots v_{|V(G)|}$. The branches at level i corresponds to including (the 0-branch) or excluding (the 1-branch) the node v_i in the pattern. The pattern along the 0-branch is the same as the parent node and can be ignored. The search tree can be constructed in this manner till we examine the node $v_{V(G)}$ at level $|V(G)|$. Figure 12.5 shows the search tree corresponding to the DFG in Fig. 12.4. The shaded regions represent the pruned design space.

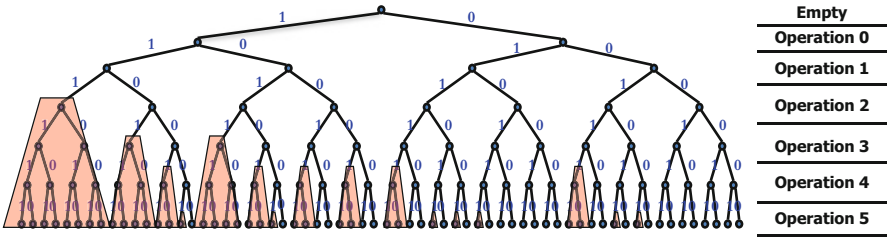


Fig. 12.5 Search tree for pattern enumeration

As mentioned earlier, the complexity of the search is reduced by employing the convexity and the output constraint. Suppose that the output constraint has been violated at a pattern. Then adding the nodes that appear later in the reverse topological order cannot possibly reduce the number of outputs for the pattern. Similarly, if the convexity constraint is violated, then it is not possible for the pattern to regain feasibility by adding the nodes that appear later in the reverse topological order. Therefore, when we reach a search tree node where either the convexity constraint or the output constraint is violated, the subtree rooted at that node can be completely eliminated from the search space.

Clearly, the search-tree-based exhaustive algorithm can prune based on the output constraint, but it cannot prune based on the input constraint. Each pattern simply needs to be checked for violation of the input constraint. Later, Pozzi et al. [55] extended the algorithm in [7] to include a simple input check. This input check is based on the observation that if a source node or an invalid node is the input to a pattern, then it is a permanent input and cannot be absorbed by adding in additional nodes to the pattern. Moreover, if an ancestor node of a pattern has been considered and excluded (0-branch), then the ancestor node also becomes a permanent input for the pattern. So if a pattern at a node in the search tree violates the input constraint based on the permanent inputs, then the subtree rooted at that node can also be eliminated. This new pruning criteria reduces the search space of the algorithm further.

12.3.3.2 Hierarchical Algorithm

The search-tree-based enumeration algorithm generates all the patterns within a single step. In contrast, Yu and Mitra [72, 73] proposed a multi-step algorithm that proceeds to generate all the feasible patterns in a hierarchical fashion. It breaks up the pattern generation process into three steps corresponding to cone, connected MIMO, and disjoint MIMO patterns. Cones are special kinds of patterns. A cone is a rooted DAG in the data-flow graph such that either there is a path from the root node r to every other node in the cone ($downCone(r)$) or there is a path from every other node to the root node ($upCone(r)$). An $upCone(r)$ is a MISO pattern if it has only one output. For example, $\{3, 5\}$ is an upCone rooted at 3 and it is also MISO pattern, but $\{0, 1, 2\}$ is an upCone rooted at 1 although it is not a MISO pattern. The pattern $\{1, 2, 3\}$ is a downCone rooted at 3.

The first step generates upCones and downCones. Recall that a MISO pattern is a upCone with only one output node. Therefore, the first step implicitly generates all the MISO patterns. The second step combines two or more cones to generate connected MIMO patterns, and finally the third step combines two or more cones/MIMO patterns to generate disjoint MIMO patterns. The hierarchical algorithm is based on the intuition that it is advantageous to separate out connected and disjoint MIMO pattern generation. The reason is the following. On the one hand, connected MIMO pattern generation algorithm does not need to consider nodes that are far apart and have no chance of participating in a connected pattern together. Therefore, the design space is reduced considerably. On the other hand, lots of infeasible patterns are filtered out during connected pattern generation step

and are not considered subsequently during disjoint pattern generation step. Thus the separation of concerns speeds up the algorithm substantially.

Theorem 1. *Any connected MIMO pattern C with $In(C)$ input operands and $Out(C)$ output operands can be generated by combining convex upCones with at most $In(C)$ input operands or convex downCones with at most $Out(C)$ output operands.*

In other words, it is possible to generate any feasible connected MIMO patterns by combining one or more cones. For example, the pattern $\{1, 2, 3, 5\}$ in Fig. 12.4 can be generated by combining $upCone(3) = \{3, 5\}$ with $downCone(3) = \{1, 2, 3\}$. The above theorem provides a key search space reduction technique by excluding some combination of cones. Specifically, to generate all the connected MIMO patterns, the hierarchical algorithm only needs all upCones that satisfy convexity/input constraints and all downCones that satisfy convexity/output constraints. This allows the algorithm to prune aggressively.

Theorem 2. *Any connected component C_i of a feasible disjoint pattern D_i must be a feasible connected pattern.*

This theorem states that a feasible disjoint pattern can be generated from one or more feasible connected patterns. The possible combination of feasible patterns is much smaller than that of arbitrary patterns, resulting in more efficient enumeration.

12.3.4 Exhaustive Enumeration of All Maximal Convex Patterns

The I/O constraint restricts the size of the valid patterns as either the input or the output constraint gets easily violated with increasing number of nodes in a pattern. Pothineni et al. [54] first proposed to relax the I/O constraint. It is well known that the speedup potential grows with increasing number of input and output nodes for the patterns [71]. Pothineni et al. observed that the convexity constraint is immutable, along with the exclusion of the invalid nodes in a pattern. They wanted to observe the limits of performance potential without I/O constraints. Later Verma et al. [66] formally proved that it is sufficient to consider only the maximal convex patterns.

Definition 1. A speedup model is monotonic, if for any two convex patterns C_1 and C_2

$$(V(C_1) \subseteq V(C_2)) \implies (Speedup(C_1) \leq Speedup(C_2))$$

Let $SW\_latency(C)$ and $HW\_latency(C)$ be the latency for software and hardware implementation of a pattern, respectively. Then Verma et al. [66] proved the following theorem.

Theorem 3. *The speedup model for pattern generation for RISC processor is monotonic, under the assumption that for any convex pattern* C, $SW\_latency(C) \geq HW\_latency(C)$.

The theorem indicates that under fairly weak assumptions, increasing the number of nodes in a pattern can never reduce the speedup. Consequently, the optimal pattern will also be the maximal pattern, and it is sufficient to enumerate only the maximal convex patterns.

However, the relaxed I/O constraint implies that the custom functional unit has to somehow obtain all the input and output operands from the register file. Cong et al. [17] proposed a shadow register file to overcome the limited bandwidth from the main register file. Jayaseelan and Mitra [36] leveraged the data forwarding or bypassing logic in the processor pipeline to supply additional operands to the CFU. Pozzi and Ienne [56] suggested distributing the register file accesses over multicycle, pipelined execution of the pattern in the CFU. This approach is known as I/O serialization in the literature. A number of algorithms [1, 4, 56, 66] have been proposed to appropriately schedule the I/O over multiple cycles to ensure that the convex pattern can be implemented in practice.

Pothineni et al. [54] defined the incompatibility graph as the first step toward solving the maximal convex pattern enumeration problem. Let $x \in X(G)$ be an invalid node in the DFG G. Clearly, any node $a \in Pred(x)$ cannot be involved with any node $b \in Succ(x)$ in a pattern because it will violate the convexity constraint. This is because there will be a path from a to b that involves the node x and the node x cannot be included in any pattern, violating convexity. Thus the cluster of nodes in $Pred(x)$ is incompatible with the cluster of nodes $Succ(x)$. Similar incompatibility can be defined between predecessor and successor nodes of each invalid node. For example, in Fig. 12.4, $\{0\}$ and $\{4\}$ are incompatible clusters. Then we can define the incompatibility graph as an undirected graph where there is an edge between each pair of incompatible cluster. Any convex pattern cannot include the incompatibility edges. Therefore, enumerating maximal convex subgraphs of G is equivalent to enumerating the maximal independent sets in the incompatibility graph (A set of vertices in a graph is independent if for every pair of vertices, there is no edge connecting the two. A maximal independent set is one which is not a proper subset of any independent set.). The maximal independent sets for the DFG in Fig. 12.4 are $\{0, 1, 2, 3, 5\}$ and $\{4, 1, 2, 3, 5\}$. Verma et al. [66] proved an equivalent result by defining a cluster graph, which is the complement of the incompatibility graph, and hence the maximal convex subgraphs can be enumerated by enumerating the maximal cliques in the cluster graph.

In general, this problem has exponential time complexity in the worst case, because the number of maximal independent sets of a graph with n nodes is upper bounded by $3^{n/3}$ [46]. But due to the clustering performed w.r.t. each invalid node in constructing the incompatibility graph, Atasu et al. [6] showed that the number of maximal convex patterns is $O(2^{|X(G)|})$ where $X(G)$ is the set of invalid nodes in the DFG G and can be enumerated in as many computational steps. Moreover, Reddington and Atasu [58] have proved that no polynomial-time maximal convex

pattern enumeration algorithm can exist. But by carefully choosing the order of clustering of the nodes, the enumeration can be performed quite effectively [5,6,40].

12.3.5 Enumeration of Maximum Weighted Convex Patterns

We can associate a weight with each vertex in the DFG. For example, the weight $weight(v)$ of a node v can correspond to the software latency $SW\_latency(v)$ of the operation corresponding to the node. Then the weight of a pattern C can be defined as $\sum_{v \in V(C)} weight(v)$. A pattern C is called the maximum (weighted) convex pattern if it is the maximal convex pattern with the maximum weight.

As mentioned earlier, it is not possible to design polynomial-time algorithm to enumerate all possible maximal convex patterns (as there can be exponential number of them present in a graph). But the problem of finding the maximum convex pattern is equivalent to finding the maximum independent set in the compatibility graph [58], and there exist polynomial-time solutions for this problem by further converting it into a minimum flow problem in a network.

Given a polynomial-time solution for the maximum convex pattern problem, we can design an iterative algorithm that identifies the maximum convex pattern in each iteration, removes those nodes from the DFG, and then repeats the process for the remaining nodes. Such an algorithm can cover the vertices of the DFG with a set of nonoverlapping convex patterns and has been demonstrated to generate high-quality custom instructions [5].

Recently, Giaquinta et al. [23] have studied the maximum weighted convex pattern identification problem under I/O constraint. This problem is useful when the maximal or maximum convex subgraphs might be too big to be realized in practice through I/O serialization. Including the I/O constraint from the beginning can generate feasible patterns that are implementable. At the same time, identifying maximum convex patterns under relatively large I/O constraint is more tractable than the original exhaustive enumeration of all convex patterns under I/O constraints discussed earlier. This problem requires first identifying the maximal convex patterns and then searching for the maximum weighted patterns from among this set.

12.3.6 Custom Instructions Selection

Given the set of candidate patterns, we need to first identify the identical subgraphs using graph isomorphism algorithm. All the identical subgraphs map to a single CFU or custom instruction; that is, a custom instruction has multiple instances. The execution frequencies of custom instruction instances are different and result in different performance gains. The selection process attempts to cover each original instruction in the code with zero/one custom instruction to maximize performance. Zero custom instruction covering an original instruction means that the original instruction is not included in any custom instruction. The selection of the custom instructions can be optimal or nonoptimal (heuristic). One way to optimally select

the custom instructions is by modeling it as an Integer Linear Program (ILP) and solve the ILP using an efficient ILP solver. The problem can also be solved optimally using dynamic programming or branch-and-bound methods.

12.3.6.1 Optimal Solution Using ILP

The ILP formulation presented here was originally proposed in [37] and then modified to this particular context in [37]. Let N be the number of custom instructions identified during the first step defined by $C_1 \ldots C_n$. A custom instruction C_i can have n_i different instances occurring in the program code denoted by $c_{i.1} \ldots c_{i.n_i}$. Each instance has execution frequency given by $f_{i.j}$. Let R_i be the area requirement of the custom instruction C_i and P_i be the performance gain obtained by implementing C_i in custom functional unit as opposed to software (given in number of clock cycles). The binary variable $s_{i.j}$ is equal to 1 if custom instruction instance $c_{i.j}$ is selected and 0 otherwise. The following objective function maximizes the total performance gain using custom instructions:

$$\max : \sum_{i=1}^{N} \sum_{j=1}^{n_i} (s_{i.j} \times P_i \times f_{i.j})$$

The objective function has to be optimized under the constraint that a static instruction can be covered by at most one custom instruction instance. If custom instruction instances $c_{i_1.j_1} \ldots c_{i_k.j_k}$ can all potentially cover a particular static instruction, then

$$s_{i_1.j_1} + \ldots + s_{i_k.j_k} \leq 1$$

In order to model the area constraint or the constraint on the total number of custom instructions, the variable S_i is defined. S_i is a binary variable that is equal to 1 if C_i is selected and 0 otherwise. S_i is defined in terms of $s_{i.j}$.

$$S_i = 1 \text{ if } \sum_{j=1}^{n_i} s_{i.j} > 0$$

$$= 0 \text{ otherwise}$$

However, the above equation is not a linear one. The following equivalent linear equations can model the constraint.

$$\sum_{j=1}^{n_i} s_{i.j} - U \times S_i \leq 0$$

$$\sum_{j=1}^{n_i} s_{i.j} + 1 - S_i > 0$$

where U is a large constant greater than $\max(n_i)$.

If R is the total area budget for all the CFUs, then

$$\sum_{i=1}^{N}(S_i \times R_i) \leq R$$

Similarly, if M is the constraint on the total number of custom instructions, then

$$\sum_{i=1}^{N} S_i \leq M$$

12.3.6.2 Other Approaches

As the ILP-based custom instruction selection may become computationally expensive for large number of custom instruction instances, heuristic selection algorithms are more practical. One idea is to assign priorities to the custom instruction instances. The instances are chosen starting with the highest priority one. In addition, any search heuristic such as genetic algorithm, simulated annealing, hill climbing, etc. can be applied for this problem. While most approaches consider single-objective optimization, the custom instruction selection exposes an interesting multi-objective optimization as well. Bordoloi et al. [10] proposed a polynomial-time approximation algorithm that can help the designers explore the area-performance trade-off for multi-objective optimization. The approach approximates the (potentially exponential size) set of points on the area-performance Pareto curve with only a polynomial number of points such that any point in the original Pareto curve is within ϵ distance (the value of ϵ is decided by the designer) from at least one of the selected points. Custom instruction selection problem has also been considered in the context of real-time systems [31, 45].

12.4 Run-Time Customization

Application-specific processor design as presented so far is quite promising. But it has a drawback that a new application-specific processor has to be designed and fabricated for at least each application domain, if not for each application. This is because a processor customized for one application domain may fail to provide any tangible performance benefit for a different domain. Soft core processors with extensibility features synthesized in Field-Programmable Gate Arrays (FPGAs) (e.g., Altera Nios [49], Xilinx MicroBlaze [60]) somewhat mitigate this problem as the customization can be performed post-fabrication. Still, customizable soft cores suffer from lower frequency and higher energy consumption issues because the entire processor (and not just the CFUs) is implemented in FPGAs. Apart from cross-domain performance problems, extensible processors are also limited by the amount of silicon available for the implementation of the CFUs. As embedded systems progress toward highly complex and dynamic applications (e.g., MPEG-4

video encoder/decoder, software-defined radio), the silicon area constraint becomes a primary concern. Moreover, for highly dynamic applications that can switch between different modes (e.g., run-time selection of encryption standard) with unique custom instructions requirements, a customized processor catering to all scenarios will clearly be a suboptimal design.

Run-time adaptive application-specific processors offer a potential solution to all these problems. An adaptive custom processor can be configured at run time to change its custom instructions and the corresponding CFUs. Clearly, to achieve run-time adaptivity, the CFUs have to be implemented in some form of reconfigurable logic. But the base processor is implemented in ASIC to provide high clock frequency and better energy efficiency. As CFUs are implemented in reconfigurable logic, these extensible processors offer full flexibility to adapt (post-fabrication) the custom instructions according to the requirement of the application running on the system and even midway through the execution of the application. Such adaptive custom processors can be broadly classified into two categories:

- *Explicit Reconfigurability:* This class of processors needs full compiler or programmer support to identify the custom instructions, synthesize them, and finally cluster them into one (or more) configuration that can be switched at run time. In other words, custom instructions are generated off-line, and the application is recompiled to use these custom instructions.
- *Implicit Reconfigurability:* This class of processors does not expose the extensibility feature to the compiler or the programmer. In other words, the extensibility is completely transparent to the user. Instead, the run-time system identifies the custom instructions and synthesizes them while the application is running on the system. These systems are more complex, but may provide better performance as the decisions are taken at run time.

12.4.1 Explicit Run-Time Customization

In this subsection, we focus on extensible processors that require extensive compiler or programmer intervention to achieve run-time reconfigurability.

Programmable Instruction-Set Processor (PRISC) [57] is one of the very first architectures to introduce CFU reconfigurability. The architecture supports a set of configurations, where each configuration corresponds to a computational kernel or custom instruction. There can be only one active configuration at any point in time. However, the CFU can be reconfigured at run time to support different configurations during the execution of an application or different applications. The temporal reconfigurability gives the illusion of a large CFU as multiple configurations can be supported using the same silicon, but it comes at the cost of reconfiguration overhead.

The CFU in PRISC is called Programmable Functional Unit (PFU). The PFU however is restricted in the sense that it can support only two input operands and one output operand. Thus the PFU cannot support large custom instructions that can

potentially provide significant performance benefit. Moreover, each configuration can only include one custom instruction. This effectively restricts PRISC to use only one custom instruction per loop body because it is expensive to reconfigure within a loop body to support multiple instructions.

OneChip [35] reduces the reconfiguration overhead by allowing multiple configurations to be stored in the PFU, but only one configuration can be active at any point of time. Unfortunately, OneChip does not provide enough details regarding how the programmers can specify or design the custom instructions that will be mapped onto the PFU.

Both PRISC and OneChip allow only one custom instruction per configuration. This decision leads to high reconfiguration overhead specially if multiple custom instructions need to be supported within a computational kernel executing frequently, such as the loop body. This restriction is lifted in the next set of architecture that enables both spatial and temporal reconfiguration. That is, multiple custom instructions can be part of a single configuration. This combination of spatial and temporal reconfiguration is a powerful feature that can significantly reduce the reconfiguration overhead.

The Chimaera [70] architecture is one of the first works to consider both temporal and spatial configuration of the custom functional units. The architecture is inspired by PRISC as it tightly couples reconfigurable functional unit (RFU) with a superscalar pipeline. But a crucial difference is that Chimaera RFU can use up to nine input registers to produce the result in one destination register. The architecture, however, suffers from inadequate compiler support. The compiler can automatically map a cluster of base instructions into custom instructions. However, the Chimaera compiler lacks support for spatial and temporal reconfiguration of custom instructions so as to fully exploit run-time reconfiguration.

The Stretch S6000 [25] architecture is a commercial processor that follows this trend of spatial and temporal reconfiguration. Figure 12.6 shows the Stretch S6000 engine that incorporates Tensilica Xtensa LX dual-issue Very Long Instruction Word (VLIW) processor [39] and the Stretch Instruction-Set Extension Fabric (ISEF). The ISEF is a software-configurable data path based on programmable logic. It consists of a plane of arithmetic/logic units (AU) and a plane of multiplier units (MU) embedded and interlinked in a programmable, hierarchical routing fabric. This configurable fabric acts as a functional unit to the processor. It is built into the data path of the base processor and resides alongside other traditional functional units. The programmer-defined application-specific instructions (called extension Instructions) need to be implemented in the ISEF. One or more custom instructions are combined into a configuration, and the compiler generates multiple such configurations. When an extension instruction is issued, the processor checks if the corresponding configuration containing the extension instruction is loaded into the ISEF. If not, the configuration is automatically and transparently loaded prior to the execution of the custom instruction. ISEF provides high data bandwidth to the core processor through 128-bit wide registers. In addition, 64KB embedded RAM is included inside ISEF to store temporary results of computation. With all

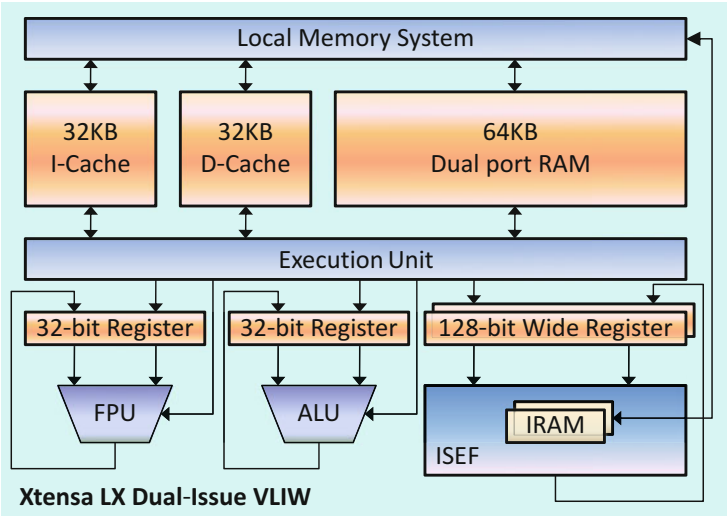


Fig. 12.6 Stretch S6000 data path [25]

these features, a single custom instruction can potentially implement a complete inner loop of the application. The Stretch compiler also fully unrolls any loop with constant iteration counts.

Most reconfigurable application-specific processors use a traditional reconfigurable fabric to implement the custom instructions or a configuration consisting of multiple custom instructions, for example, Stretch S6000 [25] architecture. This approach has the advantage of flexibility but suffers from computational inefficiency. Just-in-time customizable (JiTC) [12] architecture reconciles the conflicting demands of performance and flexibility in extensible processor. The key innovation in this architecture is a Specialized Functional Unit (SFU) tightly integrated into the processor pipeline. The SFU is a multistage accelerator that has been purpose-built to execute most common computational patterns across a range of representative applications in a single cycle. The SFU can be reconfigured on a per cycle basic to support different custom instructions in different cycles. The JiTC compiler identifies the appropriate custom instructions, generates the configuration bitstream for each such instruction to be implemented on the SFU, and replaces the selected patterns in the software binary with custom instructions. The JiTC core can thus provide near-ideal performance of an extensible processor with very little silicon area dedicated for customization. The JiTC core has recently been employed in a low-power many-core architecture [63] for wearables to enable low-cost application-specific customization at run time.

12.4.1.1 Partial Reconfiguration

So far the architecture presented requires full reconfiguration, that is, the entire fabric is reconfigured to support the next configuration. This can result in wasted

reconfiguration cost specially when there is overlap between two consecutive custom instructions. That is, only a subset of custom instructions from the current configuration should be replaced with new custom instructions. Partial reconfiguration comes to rescue in this situation as it provides the ability to reconfigure only part of the reconfigurable fabric. With partial reconfiguration, idle custom instructions can be removed to make space for the new instructions. Moreover, as only a part of the fabric is reconfigured, it saves reconfiguration cost.

Dynamic Instruction-Set Computer (DISC) [68] is one of the earlier attempts in designing an extensible processor to provide partial reconfiguration feature. DISC implements each instruction of the instruction set as an independent circuit module. Thus the individual instruction modules can be paged in and paged out onto the reconfigurable fabric in a demand-driven manner. Moreover, the circuit modules are relocatable. If needed, an existing module can be moved to a different place inside the fabric so as to create enough contiguous free space to accommodate the incoming instruction module. The drawback of DISC system is that both the base and the custom instructions are implemented in the reconfigurable logic leading to performance loss. On the other hand, the host processor remains severely underutilized as its only task is resource allocation and reconfiguration.

Extended instruction set RISC (XiRisc) [41] follows this line of development to couple a VLIW data path with a run-time reconfigurable hardware. The architecture can support four source operands and two destination operands for each custom instruction. One of the interesting developments in XiRisc is that the reconfigurable hardware can hold internal states for several cycles reducing the register pressure on the base processor. However, XiRisc did not include configuration caching leading to higher reconfiguration overhead. Also like most early reconfigurable application-specific processor, XiRisc lacked compiler support to automate the custom instruction design and reconfiguration process.

Molen [64] is an interesting polymorphic processor that incorporates an arbitrary number of reconfigurable functional units. This allows the architecture to execute two more independent custom instructions in parallel in the reconfigurable logic. To support the functionality of the reconfigurable fabric, eight custom instructions are added to the instruction set. Molen requires a new programming paradigm where general-purpose instructions and hardware descriptions of custom instructions co-exist in a program. Molen compiler automatically generates optimized binary code from applications specified in C programming language with pragma annotation for custom instructions. The architecture hides the reconfiguration cost through scheduling where the configuration corresponding to a custom instruction is pre-fetched before the instruction is executed.

12.4.1.2 Compiler Support

Compiler support is instrumental in ensuring greater adoption of application-specific processors by software designers. Unfortunately, as mentioned earlier, most of the run-time reconfigurable application-specific processors suffer from

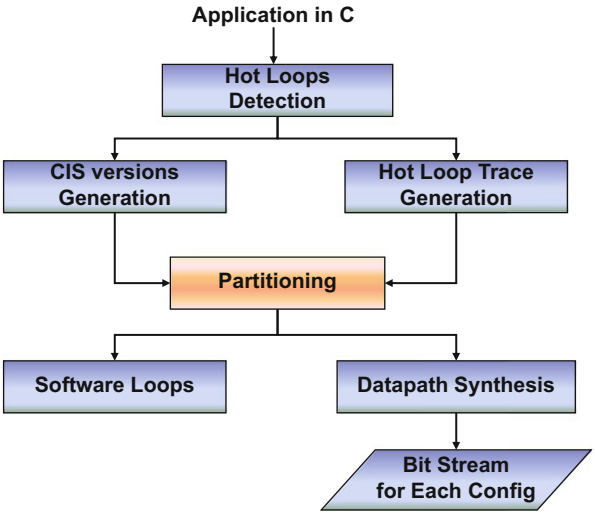


Fig. 12.7 Compiler framework for run-time adaptive extensible processors [33]

inadequate compiler assistance. The burden falls entirely on the programmer to select appropriate the custom instructions and cluster them into one or more configurations. Choosing an appropriate set of custom instructions for an application itself is a difficult problem as discussed in Sect. 12.3. Run-time reconfiguration introduces the additional complexity of the temporal and spatial partitioning of the selected custom instructions into a set of configurations.

Huynh et al. [33] developed an efficient compilation framework that takes in as input an application specified in ANSI-C and automatically selects appropriate custom instructions as well as bundles them together into one or more configurations as shown in Fig. 12.7. First, a profiling step identifies and extracts a set of compute-intensive candidate loop kernels from the application. For each candidate loop, one or more Custom Instruction-Set (CIS) versions are generated (e.g., by changing loop unrolling factor) differing in performance gain and area trade-offs. The control flows among the hot loops are captured in the form of a loop trace obtained through profiling. The hot loops with multiple CIS versions and the loop trace are fed to the partitioning algorithm that decides the appropriate CIS version and configuration for each loop. The algorithm models the temporal partitioning of the custom instructions into different configurations as a k-way graph partitioning problem. A dynamic programming-based pseudo-polynomial-time algorithm determines the spatial partitioning of the custom instructions within a configuration. The selected CIS versions to be implemented in hardware pass through a data-path synthesis tool generating the bitstream corresponding to each configuration. These bitstreams are used to configure the reconfigurable fabric at run time. The source code is modified to exploit the new custom instructions while the remaining code executes on the base processor.

12.4.2 Implicit Run-Time Customization

We now proceed to describe extensible processors that are reconfigured transparently by the run-time system.

Configurable Compute Accelerator (CCA) [15] enables transparent instruction-set customization support through a plug-and-play model that can integrate different accelerators into a predesigned and verified processor core at run time. The compiler framework comprises of static identification of subgraphs for execution on CCA [16]. This is supplemented with run-time selection of custom instructions to be synthesized to CCA. First, the program is analyzed to identify the most frequent computation patterns (custom instructions) to be mapped onto CCA. These patterns are replaced by function calls in the binary code. At run time, when the function corresponding to a custom instruction is encountered for the first time, it executes in the base processor pipeline. But, in parallel, the architecture determines the CCA configuration required for this particular custom instruction. When the same function is encountered again in the future, it can execute on the CCA using the generated configuration.

Unlike CCA that requires compiler-architecture cooperation, the WARP [42] architecture has been designed with completely transparent instruction-set customization in mind. WARP processor consists of a base core, an on-chip profiler, WARP-oriented FPGA, and an on-chip Computer-Aided Design (CAD) module. An application starts executing on the base processor. The on-chip profiler identifies the critical hot-spot kernels (loops) during the execution of the application. These kernels are then passed onto the riverside on-chip CAD (ROCCAD) tool chain through the on-chip CAD module. ROCCAD tool chain decompiles the application binary into high-level representation that is more suitable for synthesis. Next, the partitioning algorithm determines the most suitable loops to be implemented in FPGA. For the selected kernels, ROCCAD uses behavioral and register transfer level (RTL) synthesis to generate appropriate hardware descriptions. Finally, ROCCAD configures the FPGA using just-in-time FPGA compilation tools that optimizes the hardware description and performs technology mapping followed by place and route to map the hardware description onto the reconfigurable fabric. Finally, the application binary is updated to be used to kernels mapped onto the FPGAs.

A unique approach toward run-time customization is proposed in the Rotating Instruction-Set Processing Platform (RISPP) [8] architecture. RISPP introduces the notion of atoms and molecules where atom is the basic data path, while a combination of atoms creates custom instruction molecule. Atoms can be reused across different custom instruction molecules. RISPP reduces the overhead of partial reconfiguration substantially through an innovative gradual transition of the custom instructions implementation from software into hardware. At compile time, only the potential custom instructions (molecules) are identified, but these molecules are not bound to any data path in hardware. Instead, a number of possible implementation choices are available including a purely software implementation. At run time, the implementation of a molecule can gradually "upgrade" to hardware

as and when the atoms it needs become available. If no atom is available for a custom instruction, it will be executed in the base processor pipeline using the software implementation. RISPP requires fast design space exploration at run time to combine appropriate atoms and evaluate trade-offs between performance and area of the custom instructions implementations. A greedy heuristic selects the appropriate implementation for each custom instruction.

12.5 Custom Instructions for General-Purpose Computing

As mentioned earlier, the design goals for specialization in the context of general-purpose applications are somewhat different. The added custom instructions should support a large number of computations (some unknown at design time) so that only a few of them are sufficient to cover significant fraction of execution of a diverse set of applications.

An example of this approach is the specialized processors called quasi-specific cores (QsCores) proposed by Venkatesh et al. [65]. QsCores have been proposed in the context of dark silicon era where the cheap silicon area can be traded in to accommodate few QSCores. Each QsCore is an application-specific processor that can accelerate a set of computations through custom instructions. The main insight here is that there exist nearly identical code fragments within and across applications. These "similar" code fragments can be represented by a single computational pattern that is implemented as a custom instruction, thereby leading to reuse. Unlike the computation patterns we introduced before, QsCores support large hot spot containing hundreds of instructions, complex control flows, and irregular memory accesses as a single pattern as prescribed by Hameed et al. [30]. This enables the QsCores to be an order of magnitude more energy-efficient than general-purpose cores. In the general form of this architecture, a general-purpose processor core is coupled with a number of QsCores – each accelerating different computations – to create a heterogeneous tiles. The entire chip consists of a number of heterogeneous tiles, each responsible for different workloads.

A different approach is taken by Govindaraju et al. [26] where the main idea is to dynamically specialize the hardware according to the phases within an application. They introduce dynamically specialized data paths called DYnamically Specialized Execution Resource (DySER) blocks. The DySER blocks are similar to the CFUs and are integrated in the processor pipeline as additional functional units. Each block is a heterogeneous array of computational units interconnected with a circuit-switched mesh network; but unlike QsCores, there is no memory access involved within DySER block. A DySER block uses specific computational units (arithmetic and logical operations) depending on the common instruction mix of the applications. By interconnecting these operations through the network, a computational pattern can be mapped to the DySER block. The compiler partitions the application into phases, identifies the most frequently executed paths within each phase, and then maps the computations corresponding to each of these paths on

DySER blocks. Note that the identification of computational patterns as discussed in Sect. 12.3 has been restricted to within basic blocks, but Yu and Mitra had quantized the benefit of crossing basic blocks boundaries and generating custom instructions spanning multiple basic blocks along hot paths [71]. DySER reaps these benefits through a concrete architectural design and implementation. There are some similarities between DySER and coarse-grained reconfigurable arrays (CGRAs) [14]. But the main difference is that CGRAs accelerate complete loops, while DySER focuses on computation within a hot path and does not support control flow or load/store that is required to map an entire loop. The other difference lies in using computational units that are decided based on instruction mix rather than generic functional units used in CGRAs and the use of circuit-switched network. The main strength of DySER is that the same specialized hardware can accelerate different applications and diverse domains through dynamic specialization.

Gupta et al. [27] proposed a configurable coprocessor called Bundled Execution of REcurring Traces (BERET) that can leverage recurring instruction sequences in a program's execution. The instruction sequence may include intervening control instructions because of the irregularity of general-purpose code. Essentially each sequence is a hot trace that forms a loop, but is much shorter compared to the original unstructured loop body. Similar to other application-specific processor approaches for general-purpose computing, BERET also aims to support multiple applications. The architecture is based on the observation that the hot trace can be broken down into a sequence of subgraphs that can execute sequentially, while exploiting parallelism and chaining within subgraph to improve performance (as is common in any custom instruction). The concept of subgraph is called bundled execution model in this approach. The observation and insight is that many subgraph structures or patterns are common within as well as across applications. Therefore, if the architecture supports some common subgraphs, any hot trace can be mapped to a series of these subgraphs for acceleration.

12.6 Conclusions

In this chapter, we presented the current state of the art in the application-specific processor design. The application-specific processors, also known as customizable processors or specialized cores, present an exciting alternative in today's energy-constrained design space. We discussed the opportunities and challenges presented by this special class of processors and the progress made in automated design of such cores over the last decade. The renewed interest in application-specific processors for general-purpose computing and even supercomputing domain have opened up interesting new research directions, both in terms of architecture and compiler, that we hope will be pursued extensively in the coming decade.

Acknowledgments This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2014-T2-2-129.

References

1. Ahn J, Choi K (2013) Isomorphism-aware identification of custom instructions with i/o serialization. IEEE Trans Comput-Aided Des Integr Circuits Syst 32(1):34–46
2. Alippi C, Fornaciari W, Pozzi L, Sami M (1999) A dag-based design approach for reconfigurable VLIW processors. In: Proceedings of the conference on design, automation and test in Europe. ACM, p 57
3. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the spring joint computer conference, 18–20 Apr 1967. ACM, pp 483–485
4. Atasu K, Dimond RG, Mencer O, Luk W, Özturan C, Diindar G (2007) Optimizing instruction-set extensible processors under data bandwidth constraints. In: Design, automation & test in Europe conference & exhibition, DATE'07. IEEE, pp 1–6
5. Atasu K, Luk W, Mencer O, Özturan C, Dündar G (2012) Fish: fast instruction synthesis for custom processors. IEEE Trans Very Large Scale Integr (VLSI) Syst 20(1):52–65
6. Atasu K, Mencer O, Luk W, Özturan C, Dündar G (2008) Fast custom instruction identification by convex subgraph enumeration. In: International conference on application-specific systems, architectures and processors, ASAP 2008. IEEE, pp 1–6
7. Atasu K, Pozzi L, Ienne P (2003) Automatic application-specific instruction-set extensions under microarchitectural constraints. Int J Parallel Program 31(6):411–428
8. Bauer L, Shafique M, Kramer S, Henkel J (2007) Rispp: rotating instruction set processing platform. In: Proceedings of the 44th annual design automation conference. ACM, pp 791–796
9. Bonzini P, Pozzi L (2007) Polynomial-time subgraph enumeration for automated instruction set extension. In: Proceedings of the conference on design, automation and test in Europe. EDA Consortium, pp 1331–1336
10. Bordoloi UD, Huynh HP, Chakraborty S, Mitra T (2009) Evaluating design trade-offs in customizable processors. In: 46th ACM/IEEE design automation conference, DAC'09. IEEE, pp 244–249
11. Borkar S, Chien AA (2011) The future of microprocessors. Commun ACM 54(5):67–77
12. Chen L, Tarango J, Mitra T, Brisk P (2013) A just-in-time customizable processor. In: 2013 IEEE/ACM international conference on computer-aided design (ICCAD). IEEE, pp 524–531
13. Chen X, Maskell DL, Sun Y (2007) Fast identification of custom instructions for extensible processors. IEEE Trans Comput-Aided Des Integr Circuits Syst 26(2):359–368
14. Choi K (2011) Coarse-grained reconfigurable array: architecture and application mapping. IPSJ Trans Syst LSI Des Methodol 4:31–46
15. Clark N, Blome J, Chu M, Mahlke S, Biles S, Flautner K (2005) An architecture framework for transparent instruction set customization in embedded processors. In: Proceedings of the 32nd international symposium on computer architecture (ISCA'05). IEEE Computer Society, pp 272–283
16. Clark N, Kudlur M, Park H, Mahlke S, Flautner K (2004) Application-specific processing on a general-purpose core via transparent instruction set customization. In: 37th international symposium on microarchitecture, MICRO-37 2004. IEEE, pp 30–40
17. Cong J, Fan Y, Han G, Jagannathan A, Reinman G, Zhang Z (2005) Instruction set extension with shadow registers for configurable processors. In: Proceedings of the 2005 ACM/SIGDA 13th international symposium on field-programmable gate arrays. ACM, pp 99–106
18. Cong J, Fan Y, Han G, Zhang Z (2004) Application-specific instruction generation for configurable processor architectures. In: Proceedings of the 2004 ACM/SIGDA 12th international symposium on field programmable gate arrays. ACM, pp 183–189
19. Dennard RH, Gaensslen FH, Rideout VL, Bassous E, LeBlanc AR (1974) Design of Ion-implanted MOSFET's with very small physical dimensions. IEEE J Solid-State Circuits 9(5):256–268

20. Dubach C, Jones T, O'Boyle M (2007) Microarchitectural design space exploration using an architecture-centric approach. In: Proceedings of the 40th annual IEEE/ACM international symposium on microarchitecture. IEEE Computer Society, pp 262–271
21. Esmaeilzadeh H, Blem E, St Amant R, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. In: International symposium on computer architecture (ISCA)
22. Geer D (2005) Chip makers turn to multicore processors. Computer 38(5):11–13
23. Giaquinta E, Mishra A, Pozzi L (2015) Maximum convex subgraphs under i/o constraint for automatic identification of custom instructions. IEEE Trans Comput-Aided Des Integr Circuits Syst 34(3):483–494
24. Gonzalez RE (2000) Xtensa: a configurable and extensible processor. IEEE Micro 20(2):60–70
25. Gonzalez RE (2006) A software-configurable processor architecture. IEEE Micro 26(5):42–51
26. Govindaraju V, Ho CH, Sankaralingam K (2011) Dynamically specialized datapaths for energy efficient computing. In: 2011 IEEE 17th international symposium on high performance computer architecture (HPCA). IEEE, pp 503–514
27. Gupta S, Feng S, Ansari A, Mahlke S, August D (2011) Bundled execution of recurring traces for energy-efficient general purpose processing. In: Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture. ACM, pp 12–23
28. Gutin G, Johnstone A, Reddington J, Scott E, Yeo A (2012) An algorithm for finding input–output constrained convex sets in an acyclic digraph. J Discret Algorithms 13:47–58
29. Halambi A, Grun P, Ganesh V, Khare A, Dutt N, Nicolau A (2008) Expression: a language for architecture exploration through compiler/simulator retargetability. In: Design, automation, and test in Europe. Springer, The Netherlands, pp 31–45
30. Hameed R, Qadeer W, Wachs M, Azizi O, Solomatnikov A, Lee BC, Richardson S, Kozyrakis C, Horowitz M (2010) Understanding sources of inefficiency in general-purpose chips. In: ACM SIGARCH computer architecture news, vol 38, no 3. ACM, pp 37–47
31. Huynh H, Mitra T (2007) Instruction-set customization for real-time embedded systems. In: Proceedings of the conference on design, automation and test in Europe. EDA Consortium, pp 1472–1477
32. Huynh HP, Mitra T (2009) Runtime adaptive extensible embedded processors–a survey. In: International workshop on embedded computer systems. Springer, Berlin/Heidelberg, pp 215–225
33. Huynh HP, Sim JE, Mitra T (2007) An efficient framework for dynamic reconfiguration of instruction-set customization. In: Proceedings of the 2007 international conference on compilers, architecture, and synthesis for embedded systems. ACM, pp 135–144
34. Ienne P, Leupers R (2006) Customizable embedded processors: design technologies and applications. Academic Press
35. Jacob JA, Chow P (1999) Memory interfacing and instruction specification for reconfigurable processors. In: Proceedings of the 1999 ACM/SIGDA seventh international symposium on field programmable gate arrays. ACM, pp 145–154
36. Jayaseelan R, Liu H, Mitra T (2006) Exploiting forwarding to improve data bandwidth of instruction-set extensions. In: Proceedings of the 43rd annual design automation conference. ACM, pp 43–48
37. Kastner R, Kaplan A, Memik SO, Bozorgzadeh E (2002) Instruction generation for hybrid reconfigurable systems. ACM Trans Des Autom Electron Syst (TODAES) 7(4):605–627
38. Kathail V, Aditya S, Schreiber R, Rau BR, Cronquist DC, Sivaraman M (2002) Pico: automatically designing custom computers. Computer 35(9):39–47
39. Leibson S (2006) Designing SOCs with configured cores: unleashing the tensilica Xtensa and diamond cores. Academic Press
40. Li T, Sun Z, Jigang W, Lu X (2009) Fast enumeration of maximal valid subgraphs for custom-instruction identification. In: Proceedings of the 2009 international conference on compilers, architecture, and synthesis for embedded systems. ACM, pp 29–36

41. Lodi A, Toma M, Campi F, Cappelli A, Canegallo R, Guerrieri R (2003) A VLIW processor with reconfigurable instruction set for embedded applications. IEEE J Solid-State Circuits 38(11):1876–1886
42. Lysecky R, Stitt G, Vahid F (2004) Warp processors. In: ACM transactions on design automation of electronic systems (TODAES), vol 11, no 3. ACM, pp 659–681
43. Merritt R (2009) ARM CTO: power surge could create 'dark silicon'. EE Times, Oct 2009.
44. Mitra T (2015) Heterogeneous multi-core architectures. Inf Media Technol 10(3):383–394
45. Mitra T, Yu P (2005) Satisfying real-time constraints with custom instructions. In: Third IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES+ ISSS'05. IEEE, pp 166–171
46. Moon JW, Moser L (1965) On cliques in graphs. Israel J Math 3(1):23–28
47. Moore GE et al (1965) Cramming more components onto integrated circuits
48. Mudge T (2000) Power: a first class design constraint for future architectures. In: International conference on high-performance computing. Springer, pp 215–224
49. Nios I (2009) Processor reference handbook
50. Palacharla S, Jouppi NP, Smith JE (1997) Complexity-effective superscalar processors. In: Proceedings of the 24th annual international symposium on computer architecture (ISCA'97), Denver. ACM, New York, pp 206–218. doi: 10.1145/264107.264201
51. Palermo G, Silvano C, Zaccaria V (2005) Multi-objective design space exploration of embedded systems. J Embed Comput 1(3):305–316
52. Pan Y (2008) Design methodologies for instruction-set extensible processors. Ph.D. thesis, National University of Singapore
53. Patterson D, Hennessy JL (2012) Computer architecture: a quantitative approach. Elsevier
54. Pothineni N, Kumar A, Paul K (2007) Application specific datapath extension with distributed i/o functional units. In: Proceedings of the 20th international conference on VLSI design, Bangalore
55. Pozzi L, Atasu K, Ienne P (2006) Exact and approximate algorithms for the extension of embedded processor instruction sets. IEEE Trans Comput-Aided Des Integr Circuits Syst 25(7):1209–1229
56. Pozzi L, Ienne P (2005) Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In: Proceedings of the 2005 international conference on compilers, architectures and synthesis for embedded systems. ACM, pp 2–10
57. Razdan R (1994) Prisc: programmable reduced instruction set computers. Ph.D. thesis, Harvard University Cambridge
58. Reddington J, Atasu K (2012) Complexity of computing convex subgraphs in custom instruction synthesis. IEEE Trans Very Large Scale Integr (VLSI) Syst 20(12): 2337–2341
59. Reddington J, Gutin G, Johnstone A, Scott E, Yeo A (2009) Better than optimal: fast identification of custom instruction candidates. In: International conference on computational science and engineering, CSE'09. vol 2. IEEE, pp 17–24
60. Rosinger HP (2004) Connecting customized ip to the microblaze soft processor using the fast simplex link (fsl) channel. Xilinx Application Note
61. Shafique M, Garg S, Mitra T, Parameswaran S, Henkel J (2014) Dark silicon as a challenge for hardware/software co-design. In: Conference on hardware/software codesign and system synthesis (CODES)
62. Shalf JM, Leland R (2015) Computing beyond moore's law. Computer 48(12):14–23
63. Tan C, Kulkarni A, Venkataramani V, Karunaratne M, Mitra T, Peh LS (2016) Locus: low-power customizable many-core architecture for wearables. In: Proceedings of the international conference on compilers, architecture, and synthesis for embedded systems (CASES)
64. Vassiliadis S, Wong S, Gaydadjiev G, Bertels K, Kuzmanov G, Panainte EM (2004) The molen polymorphic processor. IEEE Trans Comput 53(11):1363–1375
65. Venkatesh G, Sampson J, Goulding-Hotta N, Venkata SK, Taylor MB, Swanson S (2011) Qscores: trading dark silicon for scalable energy efficiency with quasi-specific cores. In: Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture. ACM, pp 163–174

66. Verma AK, Brisk P, Ienne P (2007) Rethinking custom ise identification: a new processor-agnostic method. In: Proceedings of the 2007 international conference on compilers, architecture, and synthesis for embedded systems. ACM, pp 125–134
67. Wall DW (1991) Limits of instruction-level parallelism. In: Proceedings of the fourth international conference on architectural support for programming languages and operating systems (ASPLOS IV), Santa Clara. ACM, New York, pp 176–188. doi: 10.1145/106972.106991
68. Wirthlin MJ, Hutchings BL (1995) A dynamic instruction set computer. In: IEEE symposium on FPGAs for custom computing machines. Proceedings. IEEE, pp 99–107
69. Wulf WA, McKee SA (1995) Hitting the memory wall: implications of the obvious. ACM SIGARCH Comput Archit News 23(1):20–24
70. Ye ZA, Moshovos A, Hauck S, Banerjee P (2000) CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In: ACM SIGARCH computer architecture news, vol 28, no 2. ACM, pp. 225–235
71. Yu P, Mitra T (2004) Characterizing embedded applications for instruction-set extensible processors. In: Proceedings of the 41st annual design automation conference. ACM, pp 723–728
72. Yu P, Mitra T (2004) Scalable custom instructions identification for instruction-set extensible processors. In: Proceedings of the 2004 international conference on compilers, architecture, and synthesis for embedded systems. ACM, pp 69–78
73. Yu P, Mitra T (2007) Disjoint pattern enumeration for custom instructions identification. In: International conference on field programmable logic and applications, FPL 2007. IEEE, pp 273–278

Memory Architectures

13

Preeti Ranjan Panda

Abstract

In this chapter we discuss the topic of memory organization in embedded systems and Systems-on-Chips (SoCs). We start with the simplest hardware-based systems needing registers for storage and proceed to hardware/software codesigned systems with several standard structures such as Static Random-Access Memory (SRAM) and Dynamic Random-Access Memory (DRAM). In the process, we touch upon concepts such as caches and Scratchpad Memories (SPMs). In general, the emphasis is on concepts that are more generally found in SoCs and less on general-purpose computing systems, although this distinction is not very clearly defined with respect to the memory subsystem. We touch upon implementations of these ideas in modern research and commercial scenarios. In this chapter, we also point out issues arising in the context of the memory architectures that become exported as problems to be addressed by the compiler and system designer.

Acronyms

ALU	Arithmetic-Logic Unit
ARM	Advanced Risc Machines
CGRA	Coarse Grained Reconfigurable Architecture
CPU	Central Processing Unit
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HDL	Hardware Description Language
PCM	Phase Change Memory

P.R. Panda (✉)
Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi, India
e-mail: panda@cse.iitd.ac.in

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_14

SoC	System-on-Chip
SPM	Scratchpad Memory
SRAM	Static Random-Access Memory
STT-RAM	Spin-Transfer Torque Random-Access Memory
SWC	Software Cache

Contents

13.1 Motivating the Significance of Memory

The concept of memory and storage is of fundamental importance in hardware/-software codesign; it exhibits itself in the earliest stages of system design. Let us illustrate the ideas starting with the simplest examples and then proceed to more complex systems. With increasing system complexity, we go on to understand some of the larger trade-offs and decision-making processes involved.

Figure 13.1a shows a simple specification involving arrays and loops written in a programming language or Hardware Description Language (HDL), with some details such as type declaration omitted. Figure 13.1b shows a possible fully parallel implementation when the statement is synthesized into hardware, with four

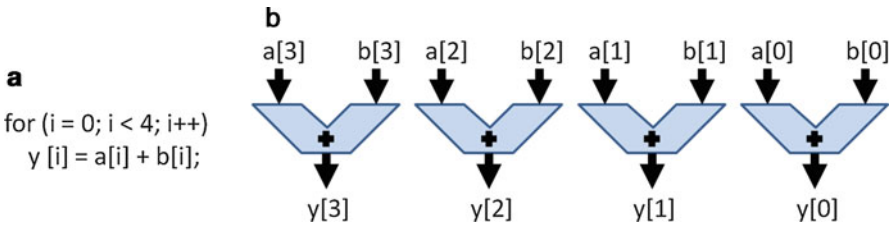


Fig. 13.1 (**a**) Code with loop and array (**b**) Hardware implementation

Arithmetic-Logic Units (ALUs). Such a specification may represent combinational logic without sequential/memory elements, as it does not involve the storage of the operands or result. Thus, in its simplest form, system implementation need not involve any memory. Note that an equivalent software implementation could consist of a sequence of addition and branch instructions, whose execution does involve registers and memory.

13.1.1 Discrete Registers

Memory elements are inferenced if we slightly modify the implementation scenario. Let us assume that there is a resource constraint of only two ALUs for implementing the specification of Fig. 13.1a. Now, we need to sequentialize the ALU computations over time so that the ALUs can be reused. This leads to registers being required in the implementation, with the computation being spread out over multiple clock cycles: $a[0]+b[0]$ and $a[2]+b[2]$ are performed in the first cycle, and $a[1]+b[1]$ and $a[3]+b[3]$ are performed in the second cycle. Since the ALU outputs have different values at different times, a more appropriate interface consists of registers connected to the ALU outputs. The select signals of the multiplexers, and load signals of the registers, would have to be generated by a small controller/FSM that asserts the right values in each cycle.

13.1.2 Organizing Registers into Register Files

The example of Fig. 13.1 was a simple instance of a specification requiring memory elements in its hardware translation. Discrete registers were sufficient for the small example. However, such an implementation does not scale well as we deal with larger amounts of data in applications. The interconnections become large and unstructured, with hard to predict area and delay behavior. For simplicity of implementation, the discrete registers are usually grouped into *register files* (RFs).

Figure 13.3 shows an alternative hardware arrangement with registers grouped into one common structure. The multiplexers, select lines, and load lines of Fig. 13.2 are now replaced with an addressing mechanism consisting of an address buses, data

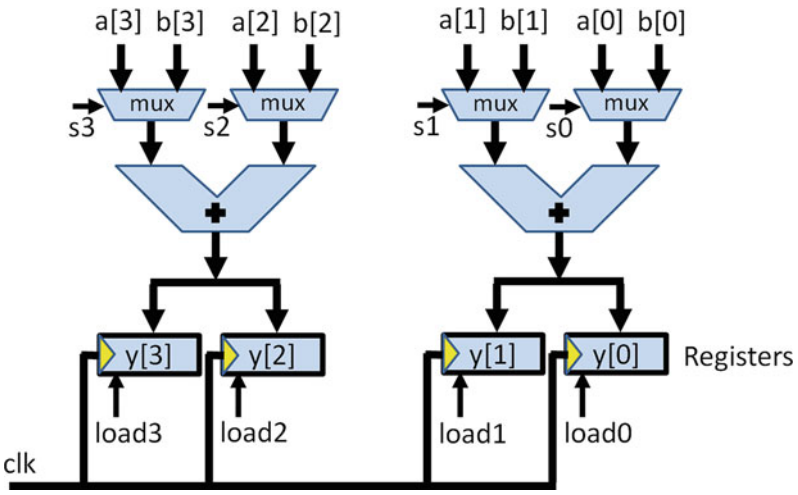
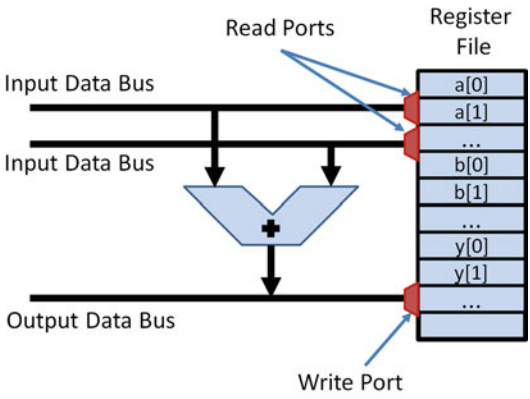


Fig. 13.2 Discrete registers in hardware implementations. Load lines are *load0* to *load3*. Mux select lines are *s0* to *s3*

Fig. 13.3 Architecture with registers grouped into register files



buses, and an internal address decoder structure that connects the data bus to the appropriate internal register. The ALU is now connected to the data buses instead of being directly connected to the registers [12]. The multiplexer and decoder structures highlighted as ports in Fig. 13.3 represent the peripheral hardware that is necessary to make the register file structure work. In other words, the register file does not consist of merely the storage cells; the multiplexing and decoding circuits do represent an area and power overhead here also, but the structure is more regular than in the discrete register case.

This abstraction of individual registers into a larger register file structure represents a fundamental trade-off involving memories in system design. The aggregation into register files is necessary for handling the complexity involving the storage and retrieval of large amounts of data in applications. One drawback of arranging the

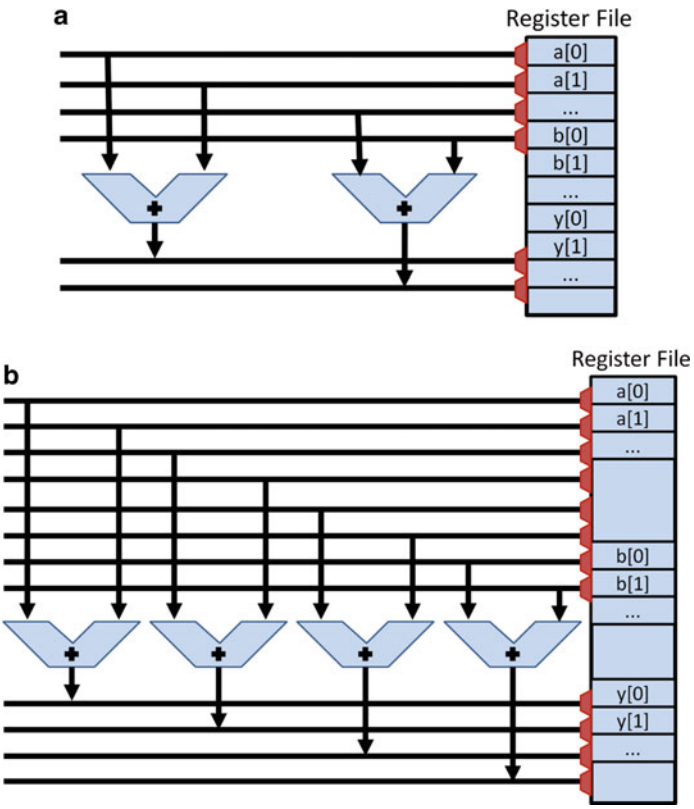


Fig. 13.4 Parallel access to register files (**a**) 6-port register file (**b**) 12-port register file

data in this way is that we have lost the ability to simultaneously access all the data, since the data bus can carry only one piece of data at any time. This leads to sequentialization of data accesses, which could impact performance severely.

To work around the sequential access problem, register files are organized to have multiple *ports*, each port consisting of an independent set of address and data buses, and control signals to indicate the operation (read, write, etc.). Figure 13.4a shows an architecture with two ALUs and a register file. In order to keep the ALUs busy, we should be able to read four operands simultaneously and also write two results back to the register file. This imposes a requirement for a total of six register file ports.

Extending the architecture to permit four simultaneous ALU operations, we observe from Fig. 13.4b that twelve ports are needed in the register file. Larger number of ports has the associated overhead of larger area, access times, and power dissipation in the register file. The peripheral multiplexing and decoding circuit increases correspondingly with the increased ports, leading to larger area and power overheads. Since the increased delays affect all register file accesses, the architecture

should be carefully chosen to reflect the requirements of the application. Alternative architectures that could be considered include splitting the register file into multiple banks, with each bank supporting a smaller number of ports. This has the advantage of faster and lower power access from the individual register file banks, while compromising on connectivity – all ALUs can no longer directly access data from all storage locations. Such trade-offs have been investigated in the context of clustered VLIW processors in general-purpose computing and also influence on-chip memory architecture choices in System-on-Chip (SoC) design. Simultaneous memory access through multiple ports also raises the possibility of access conflicts: a write request to a location through one port may be issued simultaneously with a write or read request to the same location through a different port. Such conflicts need to be resolved externally through an appropriate scheduling of access requests.

13.1.3 Packing Data into On-Chip SRAM

Registers and register files are the closest memory elements to computation hardware, making them the most quickly accessible storage. Fast access also implies an inherent constraint on their sizes – typical register files can store 16, 32, or 64 data words, which can sometimes extend to a few hundreds. When we need to process larger amounts of data, we have to incorporate other structures such as Static Random-Access Memory (SRAM). Register files and SRAMs are usually distinguished by the relative sizes and number of ports. SRAMs can accommodate hundreds of kilobytes of on-chip storage. Large SRAMs also have correspondingly fewer ports because the basic cell circuit for providing connectivity to a large number of ports does not scale well for large sizes, and the memory would incur large area, performance, and power overheads.

Figure 13.5 shows possible configurations where SRAM is integrated into SoC. In Fig. 13.5a the data bus of the SRAM is directly connected to ALUs, while in Fig. 13.5b the ALU is connected only to the register file, with the register file serving as the interface to the SRAM; data is first transferred from the SRAM to the register file before being operated upon by the ALUs. It is possible to consider discrete registers also here, instead of register files. Being denser, the SRAMs can store more data per unit area than the register files. However, the larger SRAMs also exhibit longer access times, which leads to a memory hierarchy as a natural architectural choice.

13.1.4 Denser Memories: Main Memory and Disk

The need for larger capacities in data storage leads to the incorporation of other memory structures as part of the hierarchy. Main memory and disks are the next architectural components that complete the hierarchy, with higher capacity and correspondingly higher access times. Modern main memories are usually implemented using Dynamic Random-Access Memory (DRAM), although other

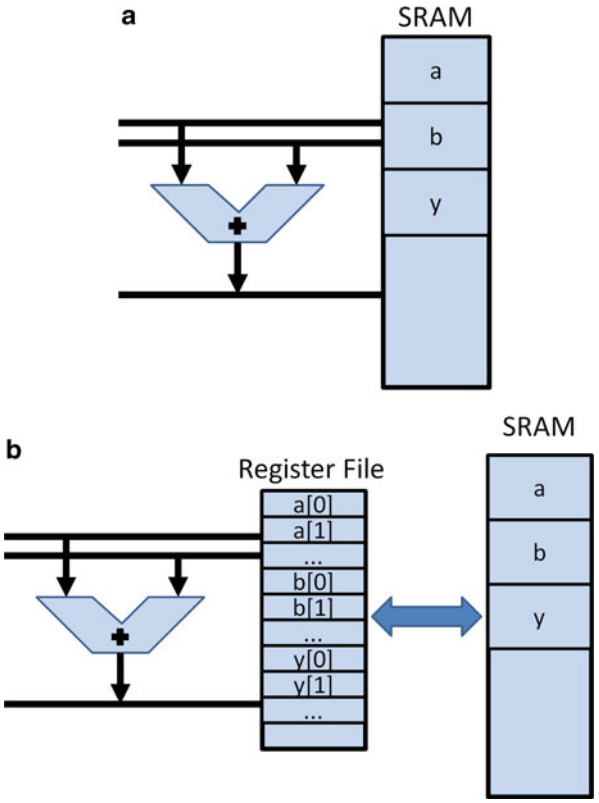


Fig. 13.5 (**a**) Data in SRAM (**b**) Hierarchically arranged register file and SRAM

memory technologies such as Phase Change Memory (PCM) and Spin-Transfer Torque Random-Access Memory (STT-RAM) have appeared on the horizon recently [14, 19, 21, 36].

The essential difference between register files and SRAM on one hand, and DRAM on the other, is that in the former, data is stored as long as the cells are powered on, whereas in DRAM the data, which is stored in the form of charge on capacitors, is lost over a period of time due to leakage of charge from the capacitors. To ensure storage for longer periods, the DRAM cells need to be refreshed at intervals. Note that *nonvolatile* memory technologies such as PCM and STT-RAM, referred to above, do not need to be refreshed in this way. However, they have other associated issues, which are discussed in more detail in ▸ Chap. 14, "Emerging and Nonvolatile Memory".

Figure 13.6 shows a simplified DRAM architecture highlighting some of the major features. The address is divided into a row address consisting of the higher-order bits and a column address consisting of the lower-order bits. A row decoder uses the row address to select a *page* from the core storage array and copy it to a buffer. A column decoder uses the column address to select the word at the right offset within the buffer and send it to the output data bus. If subsequent

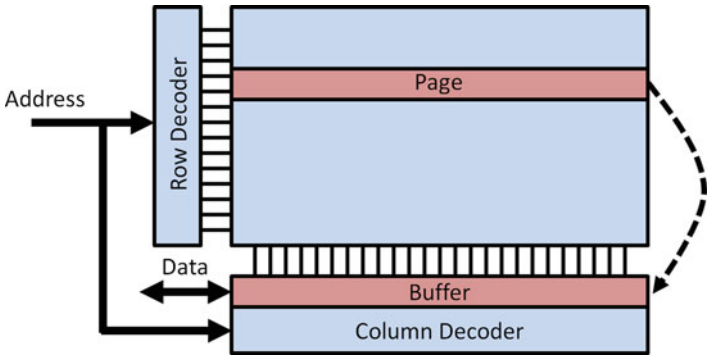


Fig. 13.6 DRAM architecture

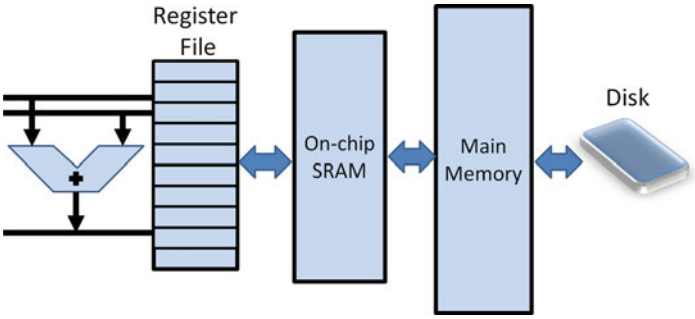


Fig. 13.7 Memory hierarchy may include disk

accesses occur to data within the same page, then we could skip the row decode stage and retrieve data directly from the buffer using only the column decode stage [30], employing what is referred to as an *open-page policy*. DRAMs have been architected around this central principle over the decades, although a large number of other structural details and management policies (including a *close-page policy* where the page is closed right after an access – a strategy that is useful when locality is weak) have been added. DRAM is generally incorporated as an off-chip component, although sometimes integrated on-chip with logic.

The final memory level in SoCs could include some form of nonvolatile storage such as solid-state disk (SSD, shown in Fig. 13.7). Occurring at a level beyond main memory, disk storage is often necessary when larger amounts of data need to be stored, for longer periods of time. The underlying technology is often flash-memory-based SSD.

13.1.5 Memory Hierarchy

System designers have to make a choice between the different types of memories discussed above, with the general trend being that smaller memories are faster,

whereas larger memories are slower. The common solution is to architect the memory system as a hierarchy of memories with increasing capacities, with the smallest memory (registers and register files) located closest to the processing units and the largest memory (DRAM and disk) lying farthest. This way, the processing units fetch the data from the closest memory very fast. There is also the requirement that the performance should not be overwhelmed by excessive accesses to the large memories.

Fortunately, the important concept of *locality of reference*, an important property exhibited by normal computing algorithms, plays an important role in this decision. *Spatial locality* refers to the observation that if a memory location is accessed, locations nearby are likely to be also accessed. This derives, for example, from (non-branch) instructions being executed as sequences, located in consecutive memory locations. Similarly, arrays accessed in loops also exhibit this property. *Temporal locality* refers to the observation that if a memory location is accessed, it is likely to be accessed again in the near future. This property can be related to instruction sequences executed multiple times in a loop and also data variables such as loop indices referenced multiple times within a short span of time, once in each iteration.

The locality property provides compelling motivation to organize the memory system as a hierarchy. If frequently accessed data and instructions can be stored/-found in levels of the memory located closer to the processor, then the average memory access times would decrease, which improves performance (and also power and energy). Within this general philosophy, a large number of combinations and configurations exist, making the overall memory architecture decision a complex and challenging process in hardware/software codesign.

13.2 Memory Architectures in SoCs

In Sect. 13.1 we reviewed the major technologies used in memory subsystems. These components are used to evolve different architectural blocks that are commonly used, depending on the requirements of the application. On-chip memories are dominated by different configurations of *caches* and *scratchpad memories*.

13.2.1 Cache Memory

Cache memory is a standard architectural block in the on-chip memory hierarchy. Designed to exploit the temporal and spatial locality properties exhibited by programs and data in typical applications, caches are SRAM-based structures that attempt to retain a copy of recently accessed information so that when it is required, the data is delivered from the cache instead of further levels of the hierarchy, thereby saving time and energy. Spatial locality is exploited by prefetching a *block* or *line*

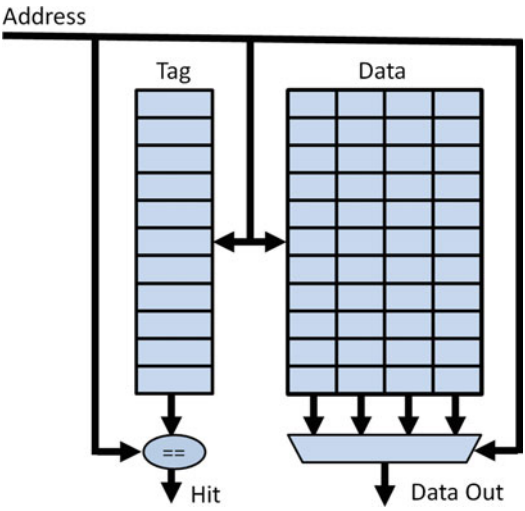


Fig. 13.8 Read operation in direct-mapped cache

of data when a single word is accessed, so that when adjacent words are accessed, they can be found in the cache. Temporal locality is exploited by implementing an appropriate *replacement policy* that attempts to retain relatively recently accessed data in the cache [12]. Caches are usually implemented using SRAM technology, but other technologies such as embedded DRAM and STT-RAM have also been explored for implementing caches [17].

Figure 13.8 shows a high-level block diagram of a *direct-mapped* cache. In this design, each address of the next memory level is mapped to one location in the cache. Since each cache level is smaller than the next level, a simple mapping function consisting of a subset of the address bits is used for determining the cache location from a given memory location. Consequently, several memory locations could map to the same cache location. When memory data is accessed, a cache line (consisting of four words from the Data memory shown in Fig. 13.8) is fetched and stored in the cache location to which the memory address maps. The higher-order address bits are also stored in the *tag* field of the cache to identify where the line came from. When a new address is presented to the cache, the higher-order address bits are compared with those stored at the corresponding location in the tag memory, and if there is a match (causing a cache *hit*), the data is delivered from the cache line itself. If there is no match (causing a cache *miss*), then the data has to be fetched from the next memory level.

The cache structure can be generalized to permit the same data to possibly reside in one of several cache locations. This permits some flexibility in the mapping and helps overcome limitations of the direct-mapped cache arising out of multiple memory addresses conflicting at the same cache location. The standard architecture for implementing this is a *set-associative* cache, with each line mapping to any of a set of locations. Figure 13.9 outlines the block diagram of a four-way set-associative cache, in which a cache line fetched from the memory can reside in one out of four

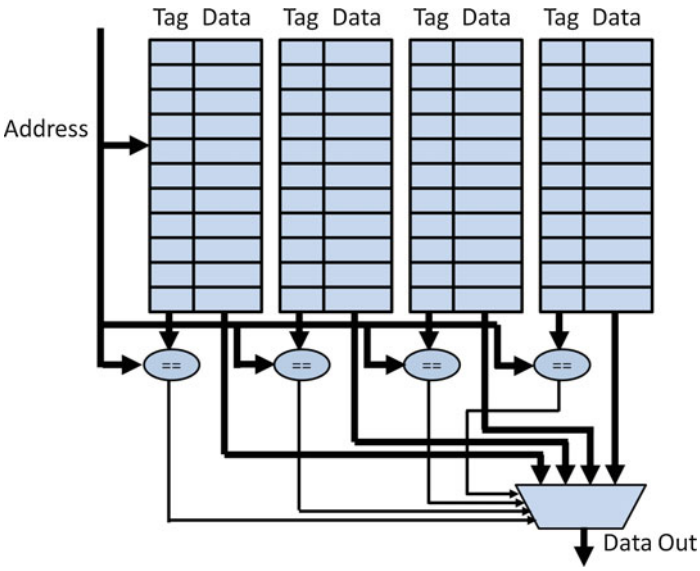


Fig. 13.9 Read operation in four-way set-associative cache

cache *ways*. The higher-order address bits are compared simultaneously with four tags, and if there is a match, a word from the corresponding way is delivered by the cache. If none of the tags match, then we have a cache miss, and a line is fetched from the next memory level. The decision of which line should be replaced in the set is taken by the replacement policy that might usually favor replacing the line that was accessed furthest in the past.

13.2.2 Scratchpad Memory

Scratchpad Memory (SPM) refers to simple on-chip memory, usually implemented with SRAM, that is directly addressable and where decisions of transfer to and from the next memory level are explicitly taken in software instead of implicitly through hardware replacement policies, as in caches [29]. Figure 13.10 shows a logical picture of the memory address map involving both SPM and cache. The caches are not visible in the address map because the cache storage decisions are not explicitly made in software. The SPM physically resides at roughly the same level as the cache, and its data is not accessed through the cache [28,32]. Although traditionally implemented in SRAM technology, other technologies such as STT-RAM are being considered for denser SPM implementation [37].

Scratchpad memory is actually simpler than caches because there is no need to include tags, comparators, implementation of replacement policies, and other control information. This makes it smaller, lower power, and more predictable,

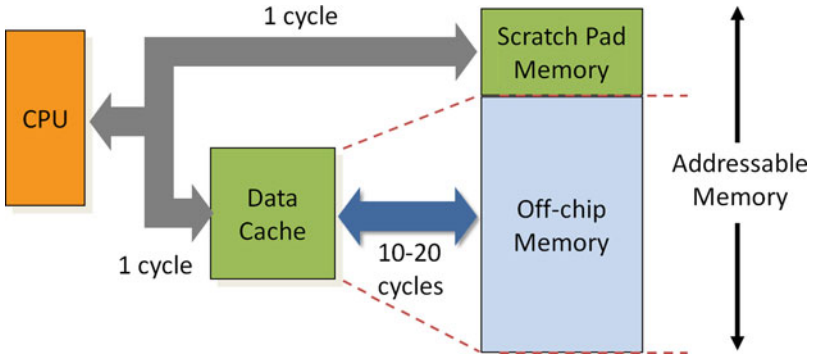


Fig. 13.10 Scratchpad memory address map

which makes it suitable for use in real-time and low-power systems. However, it does increase the work done in software (either by the developer, if done manually, or the compiler, if automated) because data transfers have to be explicitly performed in software.

Scratchpad memory could be integrated into system designs in a variety of ways, either independently or in conjunction with other memory structures. Figure 13.11 illustrates some such configurations. In the architecture of Fig. 13.11a, the local first-level memory consists of only SPM, which holds both instructions and data. The concept of a cache could still be useful, however, and a cache could be emulated within the SPM (Sect. 13.2.3). A Direct Memory Access (DMA) engine acts as the interface between the SPM and external memory. The DMA is responsible for transferring a range of memory data between different memories – in this case, between SPM and the next level. In Fig. 13.11b, the architecture supports both local SPM and hardware cache. In such systems, decisions have to be made for mapping code and data to either SPM or cache. Other variations are possible – for example, in Fig. 13.11c the local memory could be dynamically partitioned between SPM and cache.

13.2.3 Software Cache

Software Cache (SWC) refers to cache functionality being emulated in software using an SPM with no hardware cache support as the underlying structure. The tag structures discussed in Sect. 13.2.1 are still conceptually present, so they need to be separately stored in the same memory, and comparisons have to be implemented in software.

The working of a software cache is illustrated in Fig. 13.12. The SWC implementation consists of a cache line data storage and tag storage area.

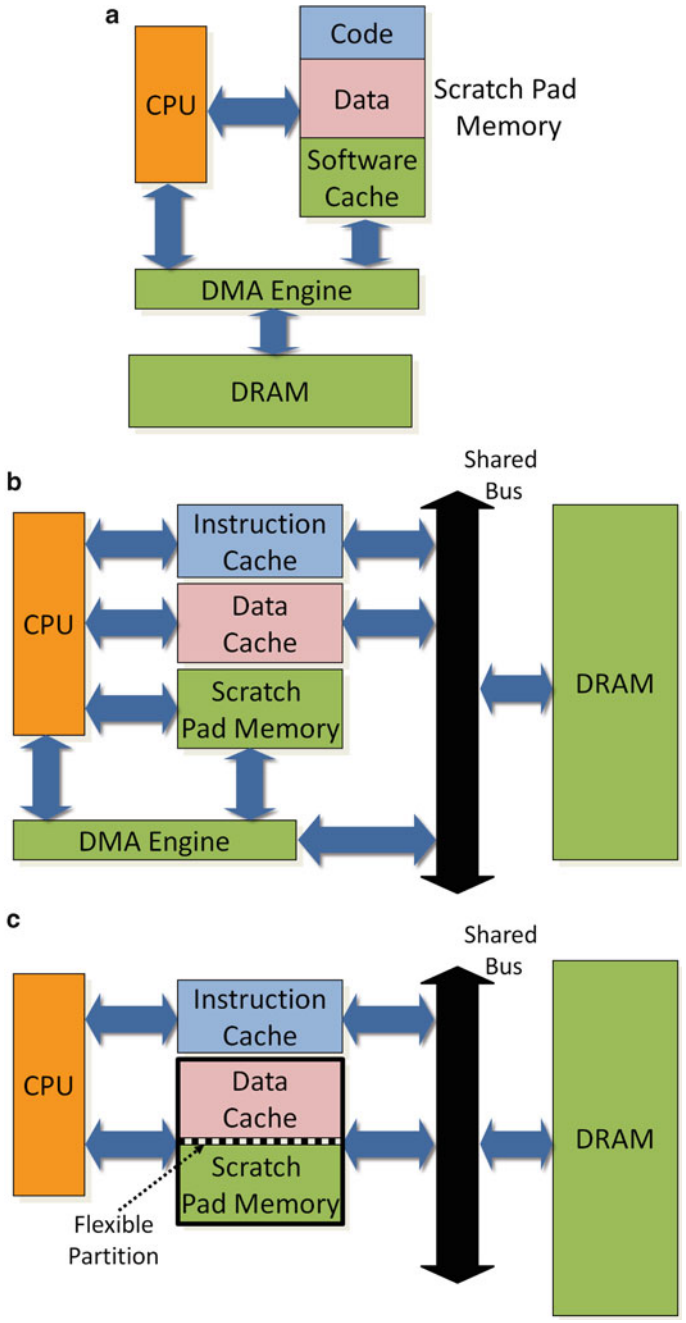


Fig. 13.11 Different architectural configurations for scratchpad memory (**a**) Local memory consists of only SPM. No hardware cache. (**b**) Local memory with both hardware cache and SPM (**c**) Dynamically configurable partition between local cache and SPM

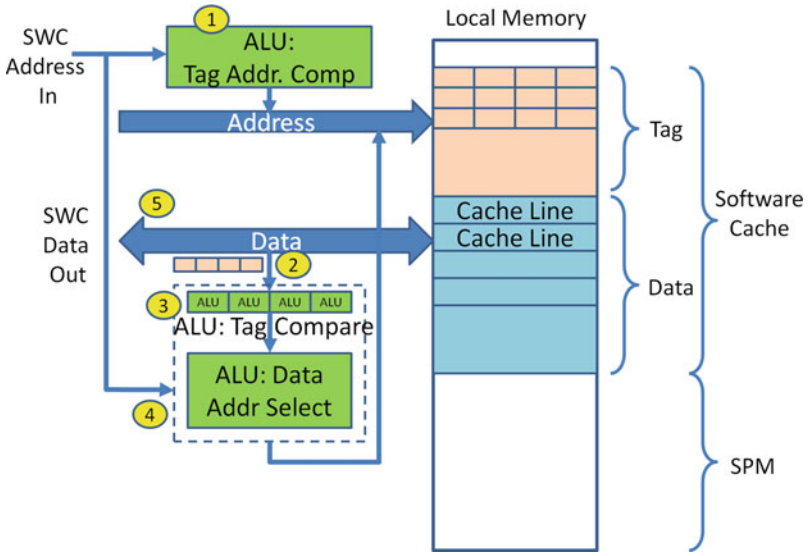


Fig. 13.12 Software caches: emulating caches in software

1. In Step 1, the index bits are extracted from the address to generate the address of the relevant tag within the SPM.
2. In Step 2, the tag is from the memory, with the four words corresponding to the four tag fields of the four-way set-associative cache.
3. In Step 3, the tags are compared against the tag bits of the address to determine a cache hit.
4. If a hit results, then the SPM address of the cache line is computed in Step 4.
5. As the final step, the data is fetched and delivered.

The emulation is relatively slow and energy inefficient compared to the hardware cache and, hence, should be judiciously used, for those data for which it is difficult to perform the static analysis required for SPM mapping [5, 7]. Efficient implementation of the software cache routines – such as a cache read resulting in a hit – used in the CELL library cause a roughly 5X performance overhead (six cycles for SPM access vs. 32 cycles for software cache access) (Sect. 13.3.2).

13.2.4 Memory in CGRA Architectures

We examine some of the on-chip memory architectures in some commercial and research SoCs and processors. Some of the designs are application specific or domain specific, while others are designed for broader applicability but still under restricted thread organization structures that do not apply to general-purpose software applications with random control structures. Such

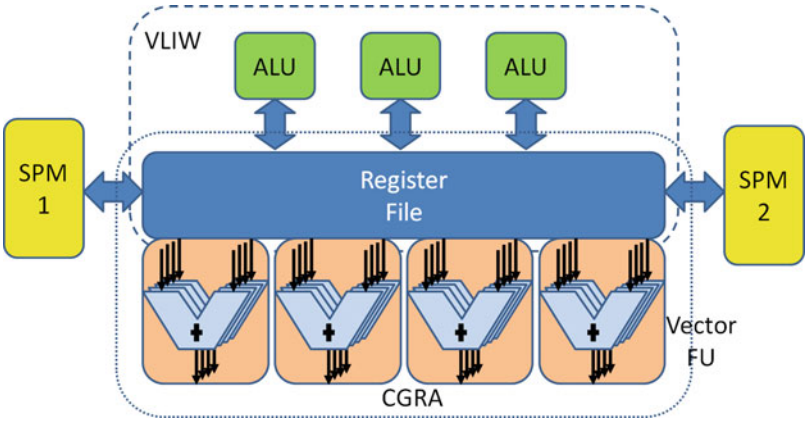


Fig. 13.13 Multiple SPMs in CGRA architecture

applications are more relevant to systems-on-a-chip rather than general-purpose computing.

The ADRES architecture [1] is a CGRA of processor cores and memories where the number of cores and memories is customizable for a specific application domain. The architecture is vector oriented, with a register file being connected to both the configurable array of vector processors (four in Fig. 13.13) and a VLIW processor that handles the control flow. In the instance of Fig. 13.13, there are two SPM blocks with wide vector interfaces to the register file. A significant responsibility lies on the system designer for utilizing such systems efficiently, which translates to challenging new problems for the compiler. The number of Central Processing Unit (CPU) cores and SPM instances has to be decided after a careful evaluation of the system throughput requirements and energy constraints. The compiler support also needs to accommodate the specialized architectures so that the application can benefit from it.

13.2.5 Hierarchical SPM

Figure 13.14 shows another instance of a processor system where a large number of independent SPMs are organized hierarchically into different memory levels [3]. Each individual core contains, apart from an ALU and register file, a level-1 SPM as well as hardware cache. Each block contains several such cores, along with a control processor with a level-2 SPM. Several such blocks exchange data through a shared memory in the form of a level-3 SPM. Finally, the entire chip is connected to a global shared memory implemented as a level-4 SPM. Such memory organizations have the ability to tremendously simplify multiprocessor architecture by not requiring complex cache coherence protocols but also need to rely on extensive support of sophisticated parallel programming environments and compiler analysis.

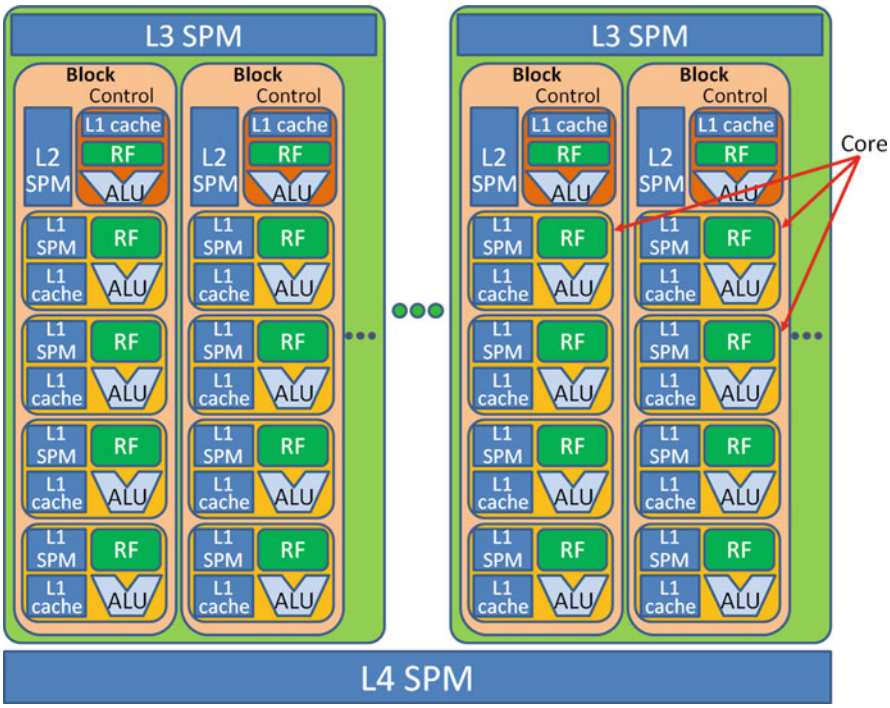


Fig. 13.14 Hierarchically organized SPM

13.3 Commercial SPM-Based Architectures

In this section we examine a few commercial architectures with interesting on-chip memory structures including scratchpad memory, in addition to conventional caches. These architectures have been used in a wide variety of applications with diverse requirements and constraints, ranging from low-power embedded systems to high-end gaming platforms and high-performance machines. We exclude general-purpose processors with conventional cache hierarchies.

13.3.1 ARM-11 Memory System

The Advanced Risc Machines (ARM) processor architecture family has been extensively used in embedded systems-on-chip. Figure 13.15 shows the local memory architecture of the ARM-11 processor, consisting of both hardware cache and SPM [2]. Data delivered from the memory subsystem could be routed from one of several sources: one of four memory banks corresponding to the four-way cache,

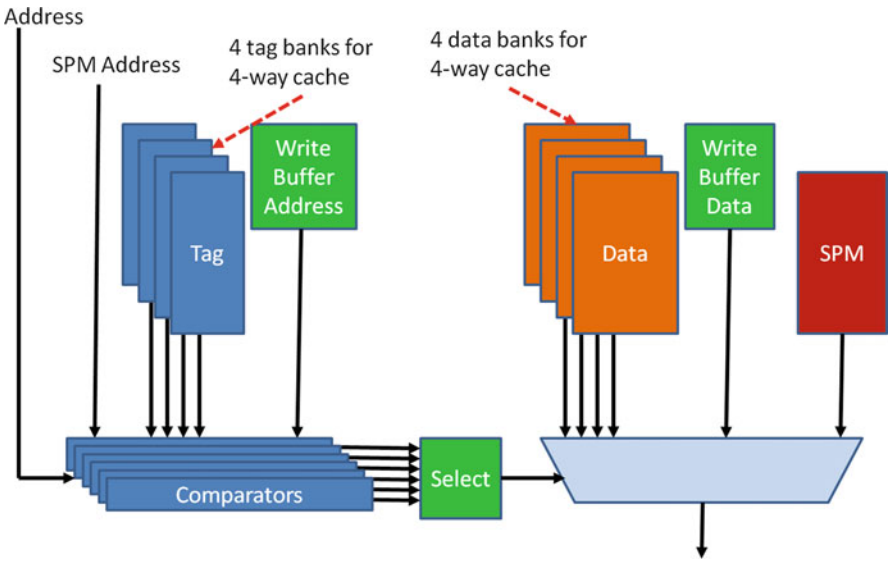


Fig. 13.15 The ARM-11 local memory architecture

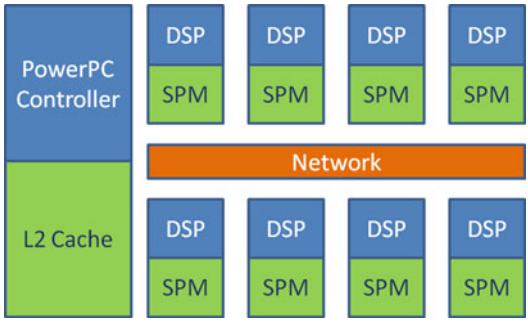


Fig. 13.16 Memory organization in the CELL processor

the SPM, and the write buffer associated with the cache. The selection of the data source is done by examining the address range, the write buffer locations, and the cache tags.

13.3.2 Local SPMs in CELL

The CELL processor [20], shown in Fig. 13.16, is a multiprocessor system consisting of eight digital signal processing engines connected over a ring network, each with a large local SPM storage intended for both instructions and data. There is no local hardware cache, and there is library support for software caches. At a higher level, a PowerPC [22] processor is used for control functions, with a regular level-2 hardware cache, which, in turn, interfaces with external DRAM.

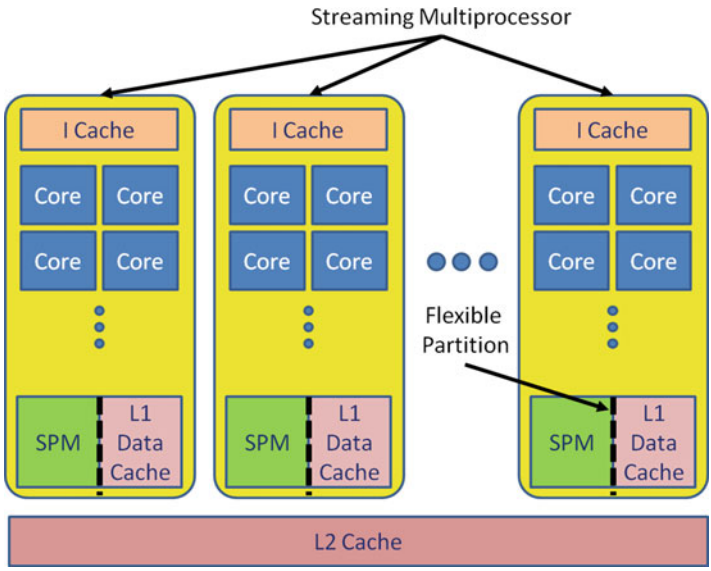


Fig. 13.17 Memory in the Fermi processor

13.3.3 Programmable First-Level Memory in Fermi

In the Fermi architecture [25] which represents the Graphics Processing Unit (GPU) class of designs (Fig. 13.17), each streaming multiprocessor consists of several processing cores with a shared data memory consisting of both SPM and hardware cache. The second-level cache is common to all the processors and interfaces with external DRAM. The architecture also supports the dynamic reconfiguration of the local memory into partitions of different SPM vs. cache sizes, depending on the requirements of the application. Although originally targeted at the graphics rendering application, architectures in this family have also been used for other applications with similar data-parallel properties, known as General-Purpose computing on Graphics Processing Units (GPGPU) applications [26].

13.4 Data Mapping and Run-Time Memory Management

The presence of advanced memory-related features requires a corresponding automated analysis functionality for exploiting the features efficiently. Conventional cache-oriented compiler analysis is sometimes applicable for these scenarios, but new mechanisms are typically necessary targeting the specialized memory structures. When data transfers are no longer automatically managed in hardware, the most fundamental problem that arises is the decision of where to map the

data from among all the memory structure choices available. Techniques have been developed to intelligently map data and instructions in SPM-based systems [6, 9, 15, 16, 38].

A major advantage of caches, from a methodology point of view, is the simplicity of their usage. Application executables that are compiled for one architecture with a given cache hierarchy also perform well for an architecture with a different cache structure. In principle, applications do not need to be recompiled and reanalyzed when the cache configuration changes in a future processor generation, although several cache-oriented analyses indeed rely on the knowledge of cache parameters. However, the other memory structures such as SPM would need a careful reanalysis of the application when the configuration is modified.

13.4.1 Tiling/Blocking

The conventional *tiling* or *blocking* optimization refers to the loop transformation where the standard iteration space covering array data is rearranged into tiles or blocks, to improve cache performance – this usually results in better temporal locality [8, 18, 33].

Figure 13.18 illustrates the tiling concept on a simple one-dimensional array, where the data is stored in main memory and needs to be fetched first into SPM for processing. Assuming no cache, the fetching and writing back of data have to be explicitly managed in software. For the code in Fig. 13.18a, the tiled version shown in Fig. 13.18b divides the array into tiles of size 100 and copies the tiles into array AA located in SPM (using the *MoveToSPM* routine). The inner loop now operates on the data in the SPM. If the data were modified, the tile would also need to be written back before proceeding to the next tile. The process is illustrated in Fig. 13.18c.

A generalization of the simple tiling concept is illustrated in Fig. 13.19a. Here, a two-dimensional array is divided into 2D tiles. In the classical tiling optimization,

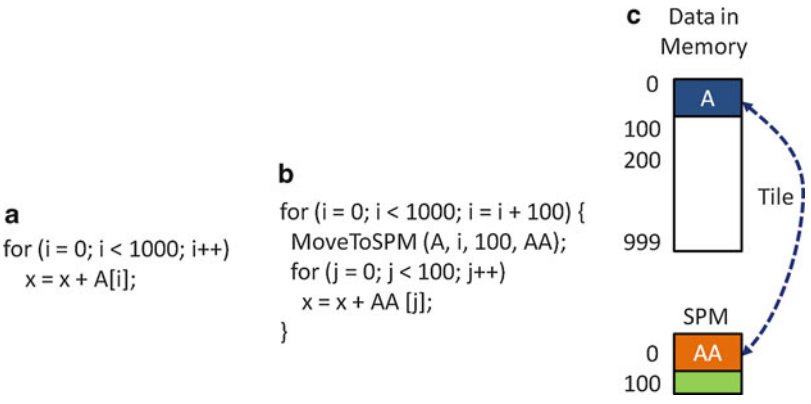


Fig. 13.18 The tiling/blocking transformation (**a**) original code (**b**) tiled code (**c**) tiling and SPM

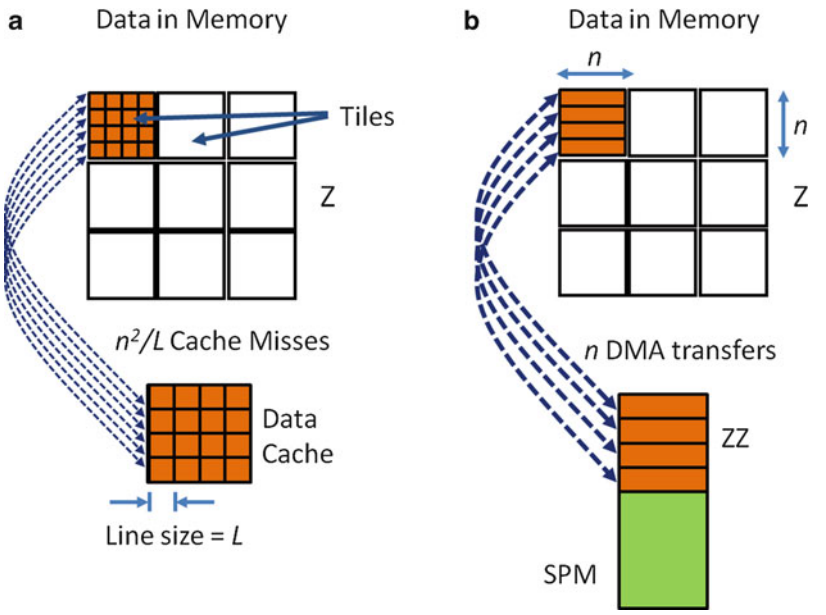


Fig. 13.19 Tiling multidimensional data (**a**) in data caches and (**b**) in scratchpad memory

the loop iteration space is modified so that the computation is first performed in one tile before proceeding to another. The tile height and width are carefully chosen such that the tile size is less than the data cache size, and also, elements within a tile exhibit minimal cache conflicts between themselves.

The tiling idea can be extended to SPM, where each tile is first fetched into the SPM for processing (Fig. 13.19b). All processing then takes place on SPM data, leading to a lower power solution because each access to the SPM is more energy efficient than the corresponding access to a hardware cache of similar capacity. Figure 13.20 shows an example of a tiled matrix multiplication targeted at SPM storage. Tiles of data are moved into the SPM using the *READ_TILE* routine before being processed. The iteration space is divided into a six-deep nested loop. This general principle is followed in SPM-based storage, where data is first fetched into the relatively small SPM, and actual processing is then performed on SPM data. The overhead of fetching data into the SPM is usually overcome by the energy-efficient accesses to the SPM.

13.4.2 Reducing Conflicts

A slightly different example of data mapping and partitioning is shown in Fig. 13.21. The array access pattern for the code shown in Fig. 13.21 is illustrated in Fig. 13.21b for the first two iterations of the j-loop. We observe that the *mask* array is small and

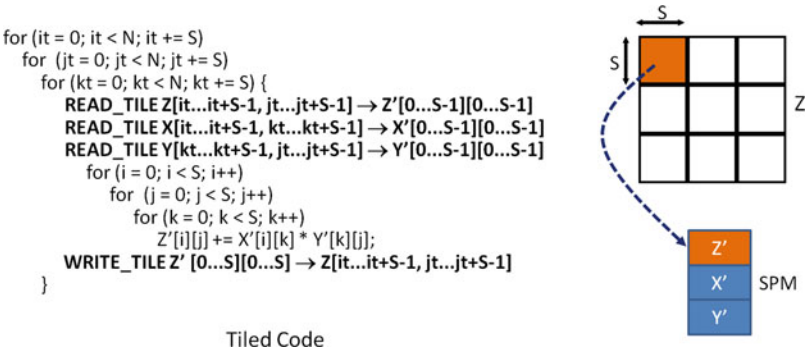


Fig. 13.20 Tiled matrix multiplication

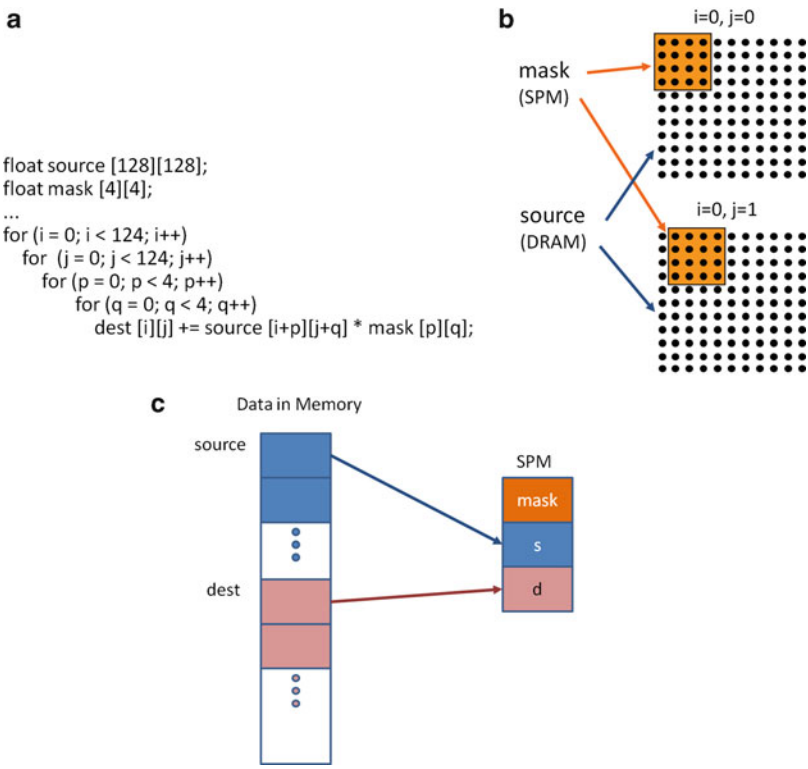


Fig. 13.21 (**a**) Convolution code (**b**) Data accesses in the first two iterations (**c**) SPM mapping

is accessed a large number of times. In comparison, the *source* array is large with the accesses exhibiting good spatial locality. It is possible that the two arrays would conflict in the cache if the cache is small. A good data mapping decision would be

to store the *mask* array in SPM and access *source* through the cache [31]. Another strategy illustrated in Fig. 13.21c is to fetch tiles from *source* and *dest* arrays into SPM for processing.

13.5 Comparing Cache and Scratchpad Memory

The quality of memory mapping decisions is closely related to the appropriate modeling of the cache and SPM parameters. Suitable high-level abstractions are necessary so that the impact of mapping decisions can be quickly evaluated. In this section we present a comparison of caches and SPM with respect to the area and energy parameters. There is no explicit access time comparison because these are similar for both; cache access times are dominated by the data array access time, which is also present in SPM.

13.5.1 Area Comparison

Caches are associated with an area overhead compared to SPM because of the tag array that is used for managing the cache. Figure 13.22 shows a comparison of the areas of SPM and a direct-mapped cache, for different memory capacities ranging from 8 to 128 KB, assuming a 32-byte line size and 32 nm process technology using the popular CACTI cache modeling tool [23]. For the same capacity, we observe a 10–32% additional area for the cache, compared to SPM.

13.5.2 Energy Comparison

We compare the energy dissipation of caches and SPM by first outlining a simple energy model of the memories expressed in terms of the different components. Since

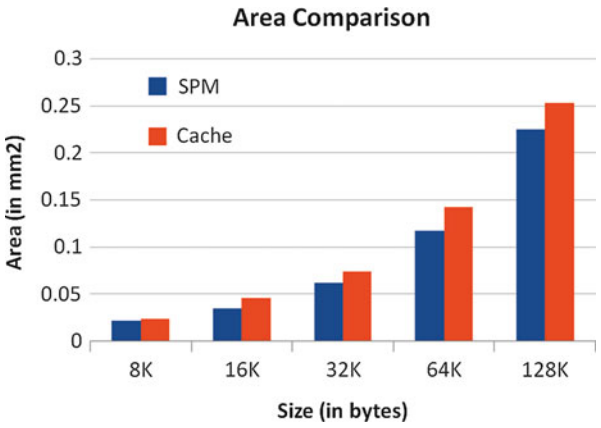


Fig. 13.22 Comparison of SPM and cache areas for different memory capacities

the mapping process may also result in conflict misses in the cache, we also study the variation of the memory energy with the conflict miss ratio, in addition to standard parameters such as memory capacity.

13.5.2.1 Energy Model for Tiled Execution

Table 13.1 shows a simplified model of the dynamic energy dissipation components of cache-based and SPM-based systems, when an $O(n^3)$ algorithm such as matrix multiplication is executed with an $n \times n$ tile. Column 1 lists the energy components in the memories.

- The computation remains identical in both cases, so the associated energy E_{Comp} is assumed to be equal.
- The dynamic energy dissipated during each memory accesses is higher in caches because of the additional energy $E_{Dyn-Tag}$ dissipated in the tag array, apart from that in the data array ($E_{Dyn-Data}$). The SPM's dynamic memory energy is limited to the data array energy $E_{Dyn-Data}$. To generate the total dynamic memory energy, the per-access energy values are multiplied by n^3, which is assumed to be the number of memory accesses required for the tile's processing.
- When the memories are idle, leakage energy is dissipated. The common data array causes a leakage energy $E_{Leak-Data}$ in both SPM and cache. The cache dissipates an additional $E_{Leak-Tag}$ due to the tag array.
- Finally, the data transfer overheads need to be carefully considered; they are sensitive to specific processor and memory architectures. The simple energy expressions for these overheads given in Table 13.1 highlight an important consideration: it is usually more energy efficient to fetch larger chunks of consecutive data from the main memory, instead of smaller chunks (the chunk size is also called *burst length* for main memory accesses). For the SPM, we assume a DMA architecture in which the $n \times n$ tile is fetched into the SPM by n DMA transfers of length n each. Each DMA transfer, fetching n elements, dissipates $E_{DMA}(n)$. In contrast, the data transfers triggered by cache misses are of size L, the cache line size, which is expected to be much smaller than n.

Table 13.1 Data cache vs. scratchpad memory energy comparison for computing an $O(n^3)$ algorithm on an $n \times n$ tile. Cache line size = L. f is the conflict miss ratio

Energy component	**SPM**	Data cache	Description
Computation	E_{Comp}	E_{Comp}	Computation stays fixed
Dynamic energy	$E_{Dyn-Data} \times n^3$	$(E_{Dyn-Data} + E_{Dyn-Tag}) \times n^3$	Tag access causes extra dynamic energy in cache
Leakage energy	$E_{Leak-Data}$	$E_{Leak-Data} + E_{Leak-Tag}$	Tag array causes extra leakage energy in cache when idle
Data transfer overheads	$E_{DMA}(n) \times n$	$E_{Miss}(L) \times (\frac{n^2}{L} + n^3 f)$	Longer burst in DMA is more energy efficient. Conflicts cause extra cache misses

Since all the tile data has to be fetched to the cache, there would be an estimated $\frac{n^2}{L}$ compulsory misses, each leading to an energy dissipation of $E_{Miss}(L)$. $E_{DMA}(n)$ for transferring a tile row of n elements is expected to be smaller than $E_{Miss}(L) \times \frac{n}{L}$, the corresponding cache energy.

Capacity misses in the cache (occurring due to insufficient cache size) are avoided by choosing an appropriate tile size, but conflict misses (occurring due to limitations of the mapping function, in spite of sufficient space) may not be completely avoided and lead to an additional $n^3 f$ misses, where f is the conflict miss ratio (defined as the number of conflict misses per access).

13.5.2.2 Sensitivity to Memory Capacity

Figure 13.23 shows a comparison of the per-access read energy for SPM and a direct-mapped cache, for different memory capacities ranging from 8 to 128 KB, assuming a 32-byte line size and 32nm process technology [23]. For the same capacity, we observe a 5–10% additional access energy for the cache, compared to SPM.

13.5.2.3 Sensitivity to Conflict Misses

Let us study the impact of one of the parameters identified in Table 13.1 – the conflict misses in tiling. As observed in Table 13.1, tiling may cause overheads in the cache if the memory accesses are subject to conflict misses. What is the extent of this overhead?

Figure 13.24 plots a comparison between the dynamic energy of the SPM and data cache for different miss ratios between 0 and 7% in a 50 × 50 tile (i.e., $n = 50$). We have ignored leakage and computation energy values and have assumed a 10% extra energy due to the tag array ($E_{Dyn-Tag} = 0.1 \times E_{Dyn-Data}$); a 2X and 4X overhead for DMA (per-word) and cache miss (per word), respectively. That is, $E_{DMA}(n) = 2n \times E_{Dyn-Data}$ and $E_{Miss}(L) = 4L \times E_{Dyn-Data}$. We notice that the memory energy overhead rises significantly as the miss ratio increases and amounts to an increase of 40% over the SPM energy for a 7% conflict miss ratio.

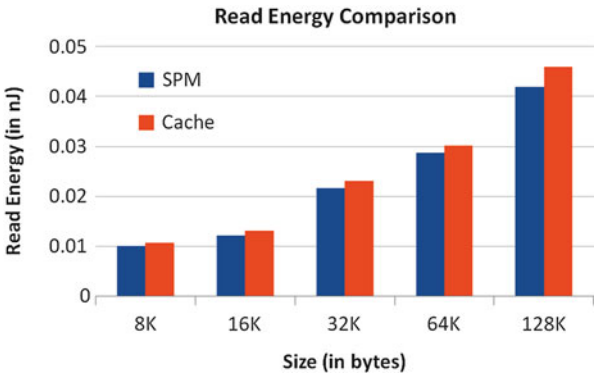


Fig. 13.23 Comparison of SPM and cache dynamic energy for different memory capacities

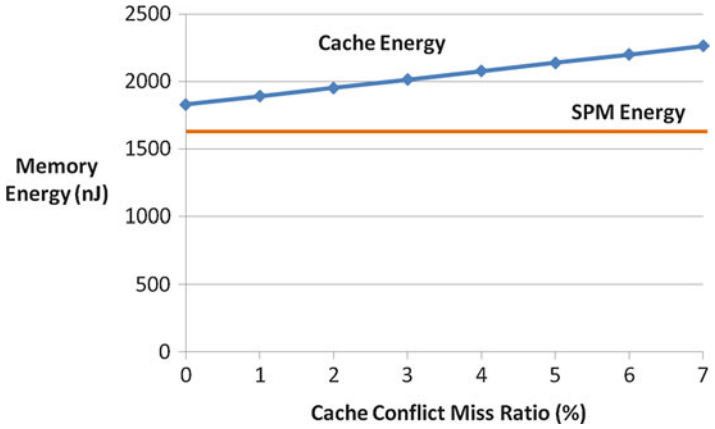


Fig. 13.24 Comparison of SPM and cache dynamic energy for different conflict miss ratios for 50×50 tile

This illustrates the importance of the appropriate mapping decision, which should ideally be implemented through a systematic procedure that estimates the benefits and cost overheads for different data mapping possibilities.

13.6 Memory Customization and Exploration

In addition to the diverse memory structures in SoC architectures, there is often the opportunity to customize the architecture for a single application or domain of applications. For example, the size and number of caches, register files, and SPMs could be customized. This process requires an exploration phase that involves iterating between architectural possibilities and compiler analysis to extract the best performance and power from each architectural instance [4, 11, 32, 35, 39]. Fast estimators are necessary to help converge on the final architecture.

13.6.1 Register File Partitioning

The register file could be an early candidate for application specific customization. The RF size could be determined based on application requirements. Further, from the application behavior, we could determine that a small set of registers need to be frequently accessed, whereas other data in the register file might not be accessed as frequently. This knowledge could be exploited by dividing the RF into two physical partitions, one smaller than the other (Fig. 13.25). If most accesses are routed to the smaller RF partition, the overall energy consumption could be smaller than the standard RF architecture where a larger RF is accessed for every register data access [24].

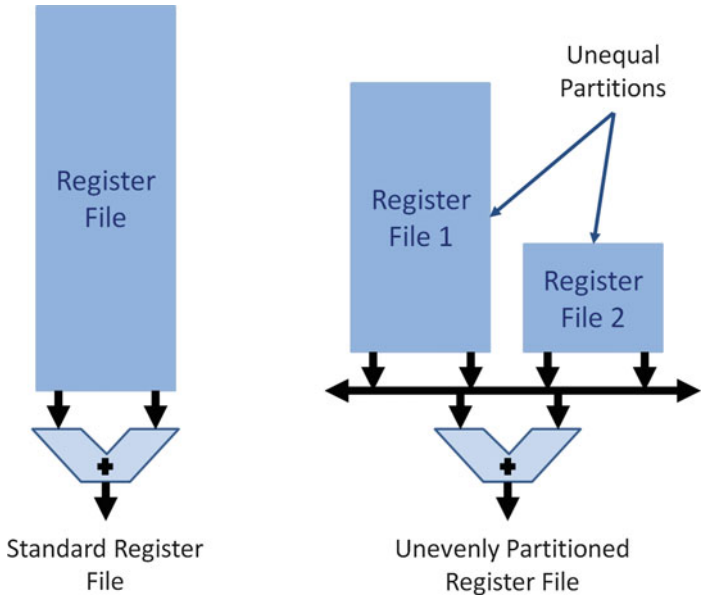


Fig. 13.25 Partitioning the register file. Uneven partitioning can lead to energy efficiency if the majority of accesses are to the smaller partition

13.6.2 Inferring Custom Memory Structures

Generalizing the register imbalance observed above, we could have a small range of memory addresses being relatively heavily accessed, leading to an analogous situation where the small range could be mapped to a small physical memory, which could lead to overall energy efficiency. Such a situation could be detected either by a compiler analysis or an execution profile of the application. This is illustrated in Fig. 13.26. The graph in Fig. 13.26a shows that a small range dominates the memory accesses. This could lead to an architectural possibility indicated in Fig. 13.26b, where this range of addresses is stored in a separate small memory. Memory accesses lying in this range could be more energy efficient because the access is made to a much smaller physical memory module [27]. The overhead of the range detection needs to be factored here. Custom memory structures could also be inferred by a data locality compiler analysis in loop nests, leading to relatively heavily accessed arrays being mapped to separate memory structures so as to improve overall energy efficiency [28].

13.6.3 Cache Customization and Reconfiguration

The presence of caches sometimes leads to opportunities for configuring the local memory in several ways. Caches themselves have a large number of parameters that

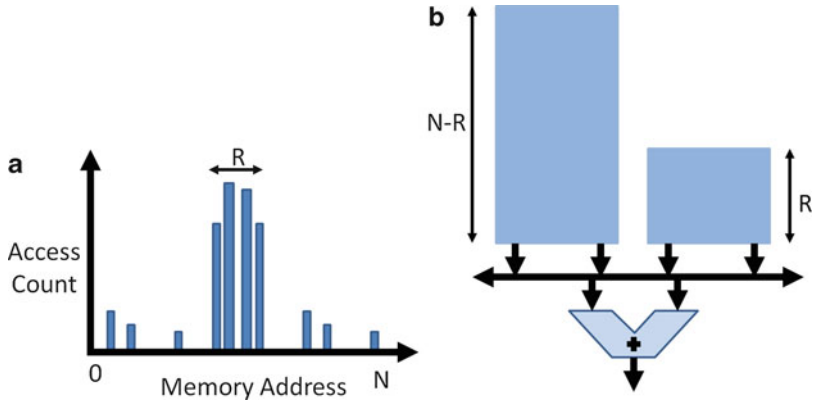


Fig. 13.26 (**a**) Profile of memory accesses (**b**) Memory architecture derived from profile

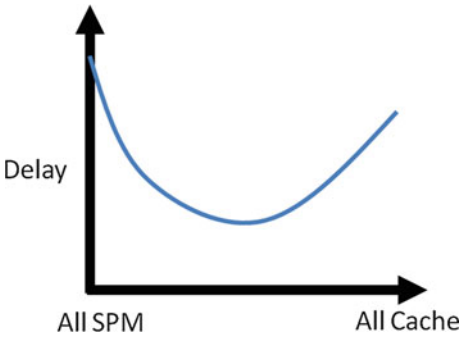


Fig. 13.27 Partitioning local memory between SPM and cache

could be tuned to the requirements of an application. Further, the coexistence of caches with other structures such as SPM expands the scope for such customization.

One exploration problem that comes up in this context is to divide a given amount of local memory space into cache and SPM. The best partitioning would be application dependent, and a compiler analysis of the application behavior would help determine the best partition. As Fig. 13.27 indicates [31], both extremes of all cache and all SPM may not be the best because different application data access patterns are suitable for different memory types. The best partition may lie somewhere in between [5, 31].

With processors such as Fermi permitting dynamic local memory reconfiguration (Sect. 13.3.3), the partitioning between cache and SPM could also be performed during the application execution, with different partitions effected during different application phases. Apart from size, possibilities also exist for dynamically reconfiguring other cache parameters such as associativity and management policy. Dynamic adjustment of cache associativity may help identify program phases where some ways can be turned off to save power. In Fig. 13.28a, the four-way cache has all four banks active at time $t1$, but two ways are turned off at $t2$ [40]. When a

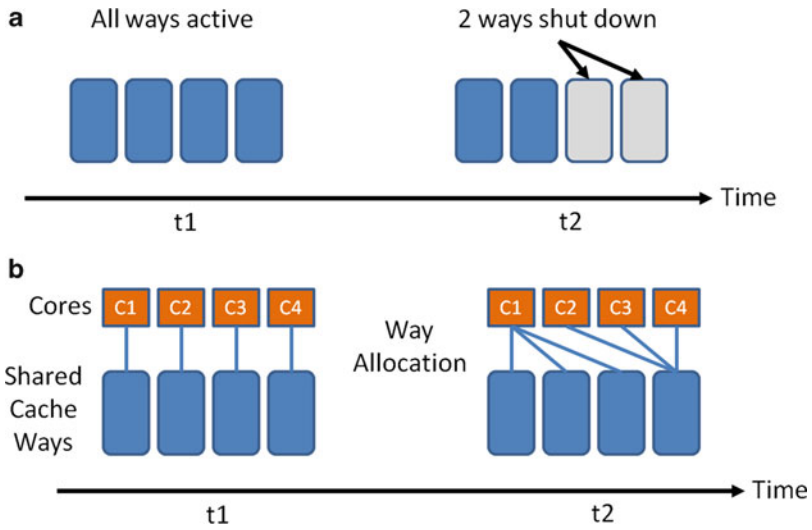


Fig. 13.28 Cache way configuration. (**a**) Ways shut down to save power (**b**) Dynamic allocation of ways to cores

cache is shared among several processor cores, an active management policy could exclusively allocate different sets of ways to different cores, with the objective of maximizing overall throughput. In Fig. 13.28b each core is allocated one exclusive way at time $t1$, but at $t2$, three ways are allocated to core $c1$, while the other way is shared among the remaining three cores [13, 34]. Such a decision could result from an analysis of the loads presented to the shared cache by the four cores. Application-specific analysis could also reveal possibilities for improving the cache mapping function [10].

13.7 Conclusions

In this chapter we reviewed some of the basics of memory architectures used in hardware/software codesign. While the principles of memory hierarchies used in general-purpose processors are relatively well defined, systems-on-chip tend to use a wide variety of different memory organizations according to the requirements of the application. Nevertheless, the architectures can be classified into a few conceptual classes such as registers, register files, caches, and scratchpad memories, instanced and networked in various ways. Codesign environments give rise to the possibility of integrating both the memory architecture as well as the application data mapping into the memory, leading to exciting technical challenges that require a fast exploration of the large number of configurations and mapping possibilities. As memory technologies continue to evolve, the problem of selecting and exploiting memory architectures continues to be relevant and expands in scope because of the

different trade-offs associated with memories with very different properties. As the technology marches forward, the integration of nonvolatile memories into system design poses very interesting new and exciting problems for the designer. This topic is discussed further in ▶ Chap. 14, "Emerging and Nonvolatile Memory".

Acknowledgments The author acknowledges Lokesh Siddhu for generating the data for memory configurations reported in Figs. 13.23 and 13.24.

References

1. Aa TV, Palkovic M, Hartmann M, Raghavan P, Dejonghe A, der Perre LV (2011) A multi-threaded coarse-grained array processor for wireless baseband. In: IEEE 9th symposium on application specific processors SASP, San Diego, 5–6 June 2011, pp 102–107
2. ARM Advanced RISC Machines Ltd (2006) ARM1136JF-S and ARM1136J-S, Technical Reference Manual, r1p3 edn
3. Carter NP, Agrawal A, Borkar S, Cledat R, David H, Dunning D, Fryman JB, Ganev I, Golliver RA, Knauerhase RC, Lethin R, Meister B, Mishra AK, Pinfold WR, Teller J, Torrellas J, Vasilache N, Venkatesh G, Xu J (2013) Runnemede: an architecture for ubiquitous high-performance computing. In: 19th IEEE international symposium on high performance computer architecture HPCA, Shenzhen, 23–27 Feb 2013, pp 198–209
4. Catthoor F, Wuytack S, De Greef E, Balasa F, Nachtergaele L, Vandecappelle A (1998) Custom memory management methodology: exploration of memory organisation for embedded multimedia system design. Kluwer Academic Publishers, Norwell, USA
5. Chakraborty P, Panda PR (2012) Integrating software caches with scratch pad memory. In: Proceedings of the 15th international conference on compilers, architecture, and synthesis for embedded systems, pp 201–210
6. Chen G, Ozturk O, Kandemir MT, Karaköy M (2006) Dynamic scratch-pad memory management for irregular array access patterns. In: Proceedings of the conference on design, automation and test in Europe DATE, Munich, 6–10 Mar 2006, pp 931–936
7. Chen T, Lin H, Zhang T (2008) Orchestrating data transfer for the CELL/BE processor. In: Proceedings of the 22nd annual international conference on supercomputing, ICS '08, pp 289–298
8. Coleman S, McKinley KS (1995) Tile size selection using cache organization and data layout. In: Proceedings of the ACM SIGPLAN'95 conference on programming language design and implementation (PLDI), pp 279–290
9. Francesco P, Marchal P, Atienza D, Benini L, Catthoor F, Mendias, JM (2004) An integrated hardware/software approach for run-time scratchpad management. In: Proceedings of the 41st annual design automation conference, DAC'04, pp 238–243
10. Givargis T (2003) Improved indexing for cache miss reduction in embedded systems. In: Proceedings of the 40th design automation conference, pp 875–880
11. Grun P, Dutt N, Nicolau A (2003) Memory architecture exploration for programmable embedded systems. Kluwer Academic Publishers, Boston
12. Hennessy JL, Patterson DA (2003) Computer architecture: a quantitative approach, 3rd edn. Morgan Kaufmann Publishers Inc., San Francisco
13. Jain R, Panda PR, Subramoney S (2016) Machine learned machines: adaptive co-optimization of caches, cores, and on-chip network. In: 2016 design, automation & test in Europe, pp 253–256
14. Jog A, Mishra AK, Xu C, Xie Y, Narayanan V, Iyer R, Das CR (2012) Cache revive: architecting volatile STT-RAM caches for enhanced performance in CMPs. In: Design automation conference (DAC), pp 243–252. doi: 10.1145/2228360.2228406

15. Kandemir MT, Ramanujam J, Irwin MJ, Vijaykrishnan N, Kadayif I, Parikh A (2001) Dynamic management of scratch-pad memory space. In: Proceedings of the 38th design automation conference, pp 690–695
16. Kandemir MT, Ramanujam J, Irwin MJ, Vijaykrishnan N, Kadayif I, Parikh A (2004) A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. IEEE Trans CAD Integr Circuits Syst 23(2):243–260
17. Komalan MP, Tenllado C, Perez JIG, Fernández FT, Catthoor F (2015) System level exploration of a STT-MRAM based level 1 data-cache. In: Proceedings of the 2015 design, automation & test in Europe conference & exhibition DATE, Grenoble, 9–13 Mar 2015, pp 1311–1316
18. Lam MS, Rothberg EE, Wolf ME (1991) The cache performance and optimizations of blocked algorithms. In: ASPLOS-IV proceedings - fourth international conference on architectural support for programming languages and operating systems, pp 63–74
19. Li H, Chen Y (2009) An overview of non-volatile memory technology and the implication for tools and architectures. In: Design, automation test in Europe conference exhibition (DATE), pp 731–736
20. Liu T, Lin H, Chen T, O'Brien JK, Shao L (2009) Dbdb: optimizing DMA transfer for the CELL BE architecture. In: Proceedings of the 23rd international conference on supercomputing, pp 36–45
21. Liu Y, Yang H, Wang Y, Wang C, Sheng X, Li S, Zhang D, Sun Y (2014) Ferroelectric nonvolatile processor design, optimization, and application. In: Xie Y (ed) Emerging memory technologies. Springer, New York, pp 289–322. doi: 10.1007/978-1-4419-9551-3_11
22. May C, Silha E, Simpson R, Warren H (1994) The PowerPC architecture: a specification for a new family of RISC processors, 2 edn. Morgan Kaufmann, San Francisco, USA
23. Muralimanohar N, Balasubramonian R, Jouppi NP (2009) CACTI6.0: A tool to model large caches. Technical Report HPL-2009-85, HP Laboratories
24. Nalluri R, Garg R, Panda PR (2007) Customization of register file banking architecture for low power. In: 20th international conference on VLSI design, pp 239–244
25. NVDIA Corporation (2009) NVIDIA's Next Generation CUDA Compute Architecture: Fermi
26. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn A, Purcell TJ (2007) A survey of general-purpose computation on graphics hardware. Comput Graphics Forum 26(1):80–113
27. Panda PR, Silpa B, Shrivastava A, Gummidipudi K (2010) Power-efficient system design. Springer, US
28. Panda PR, Catthoor F, Dutt ND, Danckaert K, Brockmeyer E, Kulkarni C, Vandecappelle A, Kjeldsberg PG (2001) Data and memory optimization techniques for embedded systems. ACM Trans Design Autom Electr Syst 6(2):149–206
29. Panda PR, Dutt ND, Nicolau A (1997) Efficient utilization of scratch-pad memory in embedded processor applications. In: European design and test conference, ED&TC '97, Paris, 17–20 Mar 1997, pp 7–11
30. Panda PR, Dutt ND, Nicolau A (1998) Incorporating DRAM access modes into high-level synthesis. IEEE Trans CAD Integr Circuits Syst 17(2):96–109
31. Panda PR, Dutt ND, Nicolau A (1999) Local memory exploration and optimization in embedded systems. IEEE Trans CAD Integr Circuits Syst 18(1):3–13
32. Panda PR, Dutt ND, Nicolau A (1999) Memory issues in embedded systems-on-chip. Kluwer Academic Publishers, Boston
33. Panda PR, Nakamura H, Dutt ND, Nicolau A (1999) Augmenting loop tiling with data alignment for improved cache performance. IEEE Trans Comput 48(2):142–149
34. Qureshi MK, Patt YN (2006) Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In: 39th annual IEEE/ACM international symposium on microarchitecture, pp 423–432
35. Ramo EP, Resano J, Mozos D, Catthoor F (2006) A configuration memory hierarchy for fast reconfiguration with reduced energy consumption overhead. In: 20th international parallel and distributed processing symposium IPDPS

36. Raoux S, Burr G, Breitwisch M, Rettner C, Chen Y, Shelby R, Salinga M, Krebs D, Chen SH, Lung H, Lam C (2008) Phase-change random access memory: a scalable technology. IBM J Res Dev 52(4.5):465–479. doi: 10.1147/rd.524.0465
37. Rodríguez G, Touriño J, Kandemir MT (2014) Volatile STT-RAM scratchpad design and data allocation for low energy. ACM Trans Archit Code Optim (TACO) 11(4):38:1–38:26
38. Steinke S, Wehmeyer L, Lee B, Marwedel P (2002) Assigning program and data objects to scratchpad for energy reduction. In: Design, automation and test in Europe, pp 409–415
39. Wuytack S, Diguet JP, Catthoor F, Man HJD (1998) Formalized methodology for data reuse: exploration for low-power hierarchical memory mappings. IEEE Trans Very Larg Scale Integr Syst 6(4):529–537
40. Zhang C, Vahid F, Yang J, Najjar W (2005) A way-halting cache for low-energy high-performance systems. ACM Trans Archit Code Optim 2(1):34–54

Emerging and Nonvolatile Memory

14

Chun Jason Xue

Abstract

In recent years, Non-Volatile Memory (NVM) technologies have emerged as candidates for future computer memory. Nonvolatility, the ability of storing information even after powered off, essentially differentiates them from traditional CMOS-based memory technologies. In addition to the nonvolatility, NVMs are also favored because of their low leakage power, high density, and comparable read speed compared with volatile memories. However, there are challenges to efficiently utilize NVMs due to the high write cost and potential endurance issues. In this chapter, we first introduce representative NVM technologies including their physical construction for data storage, as well as characteristics, and then summarize recent work aiming to exploring NVMs' characteristic to optimize their behaviors.

Acronyms

CMOS	Complementary Metal-Oxide-Semiconductor
DRAM	Dynamic Random-Access Memory
DWM	Domain Wall Memory
FeRAM	Ferro-electric Random-Access Memory
MTJ	Magnetic Tunnel Junction
NMOS	Negative-type Metal-Oxide-Semiconductor
NVM	Non-Volatile Memory
PCM	Phase Change Memory
RRAM	Resistive Random-Access Memory
SRAM	Static Random-Access Memory
STT-RAM	Spin-Transfer Torque Random-Access Memory
WL	Word Line

C.J. Xue (✉)
City University of Hong Kong, Hong Kong, Hong Kong
e-mail: jasonxue@cityu.edu.hk

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_15

Contents

14.1 Introduction

With the continuously increasing scalability, traditional CMOS-based memories are facing great challenges. Taking Dynamic Random-Access Memory (DRAM) as an example, the limited scalability and large leakage power make it undesirable for next-generation main memory. Emerging Non-Volatile Memories (NVMs) are proposed to take this challenge in future computing systems due to several promising advantages. First, NVMs have high scalability. For example, Phase Change Memory (PCM), a representative NVM, has been demonstrated in a 20 nm device prototype and is projected to scale to 9 nm, while manufacturable solutions for scaling DRAM beyond 40 nm are unknown [1, 22, 44]. Second, NVMs have a much larger storage density. In addition to scalability, the feasibility of storing multiple bits in one NVM cell further enlarges the density. Third, NVMs are nonvolatile, indicating that data will not be lost even the memory is out of power supply.

However, NVMs are commonly associated with large programming cost and possible endurance issues. As a result, new management and optimization policies should be proposed to efficiently utilize NVMs in computer and embedded systems. These policies should fully exploit the physical characteristics of NVMs, which are significantly different from volatile memories, and also tune their behaviors to fit into the memory hierarchy.

14.2 Classification of Emerging Nonvolatile Memories

Emerging nonvolatile memory includes Spin-Transfer Torque Random-Access Memory (STT-RAM), Resistive Random-Access Memory (RRAM), Domain Wall Memory (DWM), Ferro-electric Random-Access Memory (FeRAM), PCM, and so on. Various technologies differ in cell size, endurance, access speed, leakage, and dynamic power, making them fit different levels in memory hierarchy. The detailed characteristics are summarized in Table 14.1. In the following, the specific physical rationality of each representative NVM is introduced.

14.2.1 Spin-Transfer Torque Random-Access Memory

The information carrier of STT-RAM is a Magnetic Tunnel Junction (MTJ) [74]. Each MTJ contains two ferromagnetic layers and one tunnel barrier layer. One of the ferromagnetic layers (reference layer) has fixed magnetic direction, while the other one (free layer) can change its magnetic direction by an external electromagnetic field or a spin-transfer torque. If the two ferromagnetic layers have different directions, the MTJ resistance is high, indicating a "1" state; if the two layers have the same direction, the MTJ resistance is low, indicating a "0" state. The STT-RAM cell structure is composed of one Negative-type Metal-Oxide-Semiconductor (NMOS) transistor as the access device and one MTJ as the storage element, as shown in Fig. 14.1. The MTJ is connected in series with the NMOS

Table 14.1 Characteristics of different memory technologies [13, 18, 30, 35, 58, 63, 78]

	STT-RAM	RRAM	DWM	FeRAM	PCM
Cell size (F^2)	6–50	4–10	≥2	∼10	4–12
Write endurance	4×10^{12}	10^{11}	10^{16}	10^{14}	10^8–10^9
Speed (R/W)	Fast/slow	Fast/slow	Fast/slow	Fast/slow	Fast/very slow
Leakage power	Low	Low	Low	Low	Low
Dynamic energy (R/W)	Low/high	Low/high	Low/high	Low/high	Medium/high

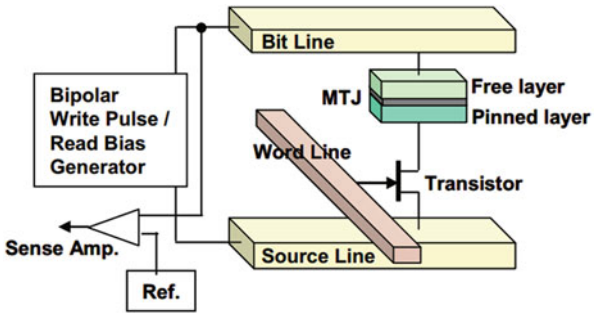


Fig. 14.1 An illustration of a "1T1J" STT-RAM cell [65]

transistor. The NMOS transistor is controlled by the wordline (WL) signal. When a write operation happens, a large positive voltage difference is established for writing "0"s or a large negative one for writing "1"s. The crystalline amplitude required to ensure a successful status reversal is called the threshold current. The current is related to the material of the tunnel barrier layer, the writing pulse duration, and the MTJ geometry [12]. STT-RAM has been widely used for designing caches due to its comparatively faster access and higher endurance than other types of NVM.

14.2.2 Resistive Random-Access Memory

Figure 14.2 shows the cell structure of RRAM. An RRAM with unipolar switching uses an insulating dielectric [26]. When a sufficiently high voltage is applied, a filament or conducting path is formed in the insulating dielectric. After this, by applying suitable voltages, the filament may be set (which leads to a low resistance) or reset (which leads to a high resistance) [35]. Compared to Static Random-Access Memory (SRAM), an RRAM cache has high density, comparable read latency, and much smaller leakage energy. However, the limitation of RRAM is its low write endurance of 10^{11} [21] and high write latency and write energy. For example, a typical 4 MB RRAM cache has a read latency of 6–8 ns and a write latency of 20–30 ns [14].

14.2.3 Domain Wall Memory

DWM works by controlling domain wall (DW) motion in ferromagnetic nanowires [58]. The ferromagnetic wire can have multiple domains which are separated by domain walls. These domains can be individually programmed to store a single bit (in the form of a magnetization direction), and thus, DWM can store multiple bits per memory cell. Logically, a DWM macro-cell appears as a tape, which stores multiple bits and can be shifted in either direction, as shown in Fig. 14.3. The challenge in using DWM is that the time consumed in accessing a bit depends on its location relative to the access port, which leads to nonuniform access latency and

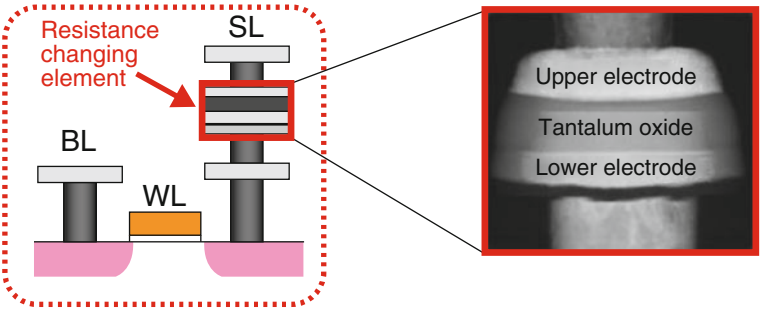


Fig. 14.2 The cell structure of RRAM [2]

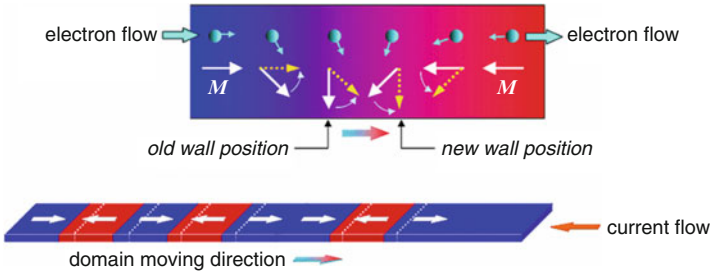


Fig. 14.3 The cell structure of DWM [77]

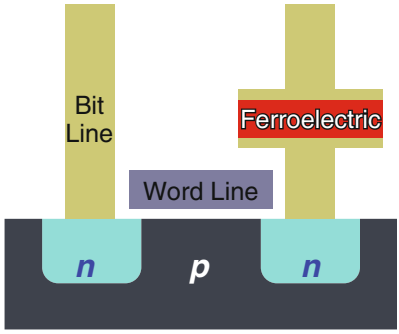


Fig. 14.4 The cell structure of FRAM with 1T-1C design [3]

makes the performance dependent on the number of shift operations required per access. Compared to other NVMs, DWM is less mature and is still in research and prototype phase.

14.2.4 Ferro-electric Random-Access Memory

A FeRAM cell contains materials with two stable polarization states, which can switch from one state to another if an external strong electric field is applied [32]. Figure 14.4 shows a popular design of a ferroelectric memory cell. This 1T-1C design uses one ferroelectric capacitor to store the information [10]. When we need to read a specific ferroelectric capacitor in a 1T-1C cell, a poled reference cell is used. By measuring the voltage/current between the capacitor and the reference cell, the stored value can be determined. The superiority of FeRAM includes nearly unlimited operation cycles, ultrashort access time and easy integration to Complementary Metal-Oxide-Semiconductor (CMOS) technology.

14.2.5 Phase Change Memory

Figure 14.5 shows the schematic of PCM memory array and cells. PCM exploits the large resistance contrast between amorphous and crystalline states in the chalcogenide alloy (GST) material. The amorphous phase tends to have high electrical resistivity, while the crystalline phase exhibits a low resistivity, sometimes

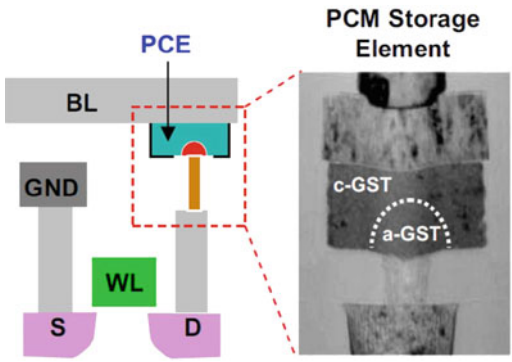


Fig. 14.5 The cell structure of PCM [37]

three or four orders of magnitude lower [44]. The amorphous and crystalline states can be transferred to each other by programming the cell into high/low resistance levels, achieved by heating the PCM cell to be with various amorphous-GST and crystalline-GST portions.

14.3 On-Chip Memory and Optimizations

In this section, the utilization of NVM as on-chip memory is summarized. First, the optimizations of on-chip caches using NVM are discussed, and then the on-chip memory for embedded systems are explored in the context of NVM.

SRAM, which has been typically used as on-chip cache, faces the problems of large leakage power and limited scalability. It is reported that, with ever-increasing required cache size, cache will occupy 90% of the chip area with tremendous fraction of chip power in future CMOS generations if consisted of SRAM [46]. Thus NVMs such as STT-RAM, RRAM, and DWM are proposed to serve as on-chip caches [35]. The write speed of NVM should be optimized when it is applied as L1 cache; the density and access energy are more important when it is used for L2 or lower-level caches [17]. Motivated by these requirements, researches have been conducted to build appropriate cache architecture, develop NVM-oriented optimizations, and revisit cache management policies to achieve high-performance low-power and dense on-chip caches.

14.3.1 STT-RAM as On-Chip Cache

STT-RAM is the mostly recommended alternative of traditional SRAM. The challenges to apply STT-RAM lie in its high programming cost and endurance. In this section, optimizations for access efficiency, endurance, as well as density are summarized.

14.3.1.1 Optimizations for Access Efficiency

Many researches focus on the latency and energy reduction of write operations in STT-RAM.

The nonvolatility of STT-RAM cells can be relaxed by cutting the planar area [52] or reducing the number of programming cycles [19, 28], by which the writes can be faster and energy efficient, however, with a shorter data retention time. The retention time should be guaranteed to be smaller than inter-write time; otherwise, additional refreshes are necessary [55]. STT-RAM cell sizing is studied for the impact on overall computing system performance, showing that different computing workloads may have conflicting expectations on memory cell sizing [64]. Leveraging MTJ device switching characteristics, the authors further propose an STT-RAM architecture design method that can make STT-RAM cache with relatively small memory cell size perform well over a wide spectrum of computing benchmarks [64]. Early write termination is proposed to save programming energy by terminating write operations once it confirmed that the data to write are the same with the old values [76]. Instead of read-before-write, this strategy senses the old data during the write process and thus does not induce latency overhead. Based on the observation that a large fraction of data written to L2 cache are "all-zero-data," flags are employed to label these words to avoid being written and read [20]. Accurate data can be constructed by unlabeled words and zero flags.

In hybrid caches consisting of both SRAM and STT-RAM, cache partitioning is explored in [49] to determine the amount of SRAM and STT-RAM ways by exploiting the performance advantage of SRAM and high density of STT-RAM. A dynamic reconfiguration scheme which determines the portion of SRAM and STT-RAM is proposed in [7] for access energy saving. Data migrations aiming to minimize the number of writes to STT-RAM can reduce the cache energy [40].

14.3.1.2 Optimizations for Endurance

Wear leveling in STT-RAM aims to balance the number of writes across different physical regions so that the cache endurance can be prolonged. The basic idea is to swap write-intensive regions with those rarely written ones. Wear leveling can be conducted with various granularites. A concept of cache color containing a number of sets is developed as the granularity for data swapping in [50]. The mapping between physical regions and cache colors is periodically remapped, with the objective of mapping the mostly written region to cache colors with the least number of writes. In a set-level wear leveling [6], cache sets are reorganized through XOR operation between changing remap register and set indexes in order to balance writes across cache sets. As an intra-set weal leveling, WriteSmoothing [36] logically divides the cache sets into multiple modules. For each module, it collectively records number of writes in each way for any of the sets. WriteSmoothing then periodically makes most frequently written ways in a module unavailable to shift the write pressure to other ways in the sets of the module. A coding scheme for STT-RAM last-level cache is proposed based on the concept of value locality in [67]. The

switching probability in cache can be reduced by swapping common patterns with limited weight codes to make writes less often as well as more uniform. This belongs to bit-level wear leveling.

In hybrid caches consisting of both SRAM and STT-RAM, similar wear leveling strategies can be modified and applied to balance writes among the whole cache space, such as data swapping between SRAM and STT-RAM [27,30]. Prementioned early write termination [76], write mode selection [19, 28], and write reductions [5,20,45] also benefit the cache endurance.

14.3.1.3 Optimizations for Density

Three-dimensional die stacking increases transistor density by vertically integrating multiple dies with a high-bandwidth, high-speed, and low-power interface [35]. Using 3D die stacking, dies of even different types can be stacked. Several researchers discuss 3D stacking of NVM caches [30, 49, 53, 54, 63]. For example, an STT-RAM cache and CMP logic can be designed as two separate dies and stacked together in a vertical manner [53]. One benefit of this is that the magnetic-related fabrication process of STT-RAM may not affect the normal CMOS logic fabrication. Three-dimensional stacking also enables shorter global interconnect, lower interconnect power consumption, and smaller footprint [35]. Cell sizing can also potentially improve the cache density by shrinking STT-RAM cells.

14.3.2 Other NVMs as On-Chip Memory

RRAM and DWM can also be applied as on-chip caches. Compared with STT-RAM, the endurance of RRAM is more serious based on the report that RRAM can withstand 10^{11} writes while STT-RAM can withstand 4×10^{12}. Thus there are researches focusing on the endurance enhancement of RRAM at the levels of both inter-set and intra-set [34,61]. Since DMW exploits the shift of access port to access data, the optimizations of DMW include hiding the shifting time [60] and reduce the shift cost [56,59].

NVMs can also be used in embedded systems as on-chip memory, such as nonvolatile flip-flops [62], Flash, and FeRAM [4]. A novel usage of NVM in embedded systems is to back up volatile program execution states upon power failures, so that it can be resumed after being recharged. In this scenario, the backup efficiency directly affects system performance and energy consumption. Nonvolatile flip-flops are connected to each volatile cell for efficient backup in [62], which is suitable for registers. Contents to back up can be reduced to achieve high performance and energy efficiency [29, 73]. Hybrid on-chip memory with Flash and FeRAM leads to a promising tradeoff between system performance and price [4].

14.4 Hybrid Main Memory and Optimizations

In this section, the utilization of hybrid PCM/DRAM as main memory is summarized. Firstly, the architecture of pure-PCM working as the main memory is developed by a few works. Then the hybrid DRAM/PCM memory architecture overview is presented. Next, we summarize the research works on two different hybrid architectures.

14.4.1 PCM as Main Memory Architecture

As emerging nonvolatile memory technologies, several researches have tried to replace DRAM with nonvolatile memory like PCM, as a candidate of the main memory. The work [23, 75] firstly developed the architecture-level studies on using PCM to implement main memory. [23] proposed to use narrow rows and multiple buffers to improve write coalescing and perform partial writes. [75] took advantage of redundant bit writes to eliminate unnecessary writes to PCM and perform dynamic memory mapping at memory controller to achieve wear-out leveling.

Based on the proposed architecture, some works have been proposed to address different issues of PCM when working as the memory, including improving endurance [23, 41, 42, 75], improving write latency by reducing the writes of bits to PCM [8, 15, 23, 66, 70, 72, 75], and preventing malicious wear-outs [48].

However, pure PCM-based main memory has many challenging issues such as slow latency and limited endurance to be used as the main memory. To overcome these limitations, the hybrid main memory is proposed instead, which is constructed using both PCM and DRAM. The following sections introduce the hybrid memory architecture and summarize the techniques used in the hybrid memory.

14.4.2 PCM/DRAM Hybrid Memory Overview

In the hybrid PCM and DRAM memory, the DRAM can help improve the slow write performance of PCM and also increases the lifetime of PCM by "absorbing" the write-intensive operations to PCM. As shown in Fig. 14.6, two different architectures have been proposed for the hybrid memory system. For the first architecture, as depicted in Fig. 14.6a, DRAM is regarded as an upper-level cache to the main memory, which only consists of PCM, to combine the short latency of DRAM with large capacity of PCM. When a memory access happens, the DRAM cache is checked firstly, and the PCM will be accessed only under the cache miss. In this architecture, the DRAM cache is invisible to the operating system. All the operations to the DRAM cache are managed by the memory controller. Qureshi et al. [42] showed that a small DRAM cache, which is 3% size of the PCM storage, can make up the latency gap between DRAM and PCM.

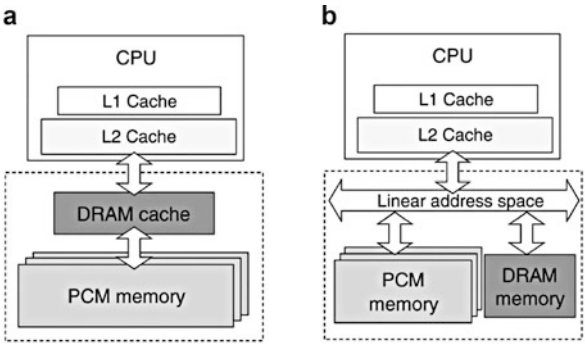


Fig. 14.6 Hybrid memory architectures consisting of both PCM and DRAM [25]. (**a**) DRAM-as-cache hybrid architecture. (**b**) Parallel hybrid memory architecture

In the second memory architecture, as shown in Fig. 14.6b, both of PCM and DRAM memory are directly connected to memory controller in parallel with each other and thus share the single memory physical space. The operating system can determine the page allocation and memory management strategy to improve the performance, energy saving, and endurance. Dhiman et al. [11] proposed the write-frequency-guided page placement policy to perform uniform wear leveling across all PCM pages, while migrating write-intensive pages to DRAM to improve performance.

In the following sections, we summarize the research works about the hybrid DRAM and PCM memory system.

14.4.3 DRAM-as-Cache Architecture

In the DRAM-as-Cache architecture, DRAM works as a cache to serve the memory requests at low latency with low energy cost and high endurance. On the other hand, PCM works as a large background main memory to exploit low standby power as well as the low costs, but the write operations should be reduced because of the limited endurance and high write latency. The key challenge in the design of such a hybrid memory system lies in the following aspects: what is the caching/paging scheme to the DRAM-PCM architecture, in terms of different objectives, including power, performance, capacity, endurance, etc., and what data should be cached in DRAM to best exploit the advantages and overcome the disadvantages of hybrid system.

14.4.3.1 Caching/Paging Schemes to the Hybrid Architecture

A series of works have proposed different caching/paging schemes to the hybrid architecture to minimize the write operations in PCM for the benefits of both PCM endurance and performance and reduce the refreshes of DRAM to explore further energy saving.

Qureshi et al. [42] firstly proposed to increase the size of main memory by using PCM as main memory and using DRAM as a cache to PCM. They developed the

Lazy-Write scheme based on the proposed memory model to minimize the write operations to PCM by only writing the pages fetched from hard disk to DRAM cache. Intra-page wear leveling is also applied in PCM to swap cache lines to achieve uniform writing of lines within pages to further improve the endurance. Experimental results show that the hybrid memory can improve 3× latency and reduce 5× page faults because of the increase in main memory size, compared to the pure-DRAM memory and achieve 3× endurance improvement compared to the pure-PCM memory.

Ferreira et al. [15] proposed write-back minimization scheme with new cache replacement policies and PCM-aware swap algorithm for wear leveling, while avoiding unnecessary writes. Zhang and Li [71] employed the PCM to implement the 3D-stacked memory systems to exploit the low standby power feature of PCM and proposed an OS-level paging scheme that takes into account the memory reference characteristics and migrated the hot-modified pages from PCM to DRAM. In this way, the lifetime degradation of PCM is alleviated, and the energy efficiency of the memory system is also improved.

To reduce the energy consumption of the hybrid system, Park et al. [38] proposed to decay the contents in DRAM. In this way, the clean, old data in DRAM are evicted, while the dirty data are written back to PCM and then evicted. Thus the evicted rows do not need refresh operations, and the energy consumption of the hybrid memory can be reduced compared to the DRAM-only memory. A long-term dirty data write-back scheme is further developed to minimize the PCM writes.

14.4.3.2 What Data Should Be Cached in DRAM

The work in [68, 69] mainly proposed an answer to the question "what data should be placed in DRAM cache." Yoon et al. observed that PCM and DRAM have the same latency to access their row buffers, while the write latency is much higher in PCM than in DRAM when accessing the content which is not stored in the row buffer. For the sake of performance and PCM endurance, the access to PCM when the row buffer miss happens should be avoided. With this observation, Yoon et al. proposed to put those rows of data that are likely to miss in the row buffer and also likely to be reused in DRAM cache. Further, their technique uses a caching policy such that the pages that are written frequently are more likely to stay in DRAM.

Meza et al. [33] propose a technique for efficiently storing and managing the metadata (such as tag, replacement-policy information, valid, and dirty bits) for data in a DRAM cache at a fine granularity. They observed that in DRAM cache storing metadata in the same row as their data can exploit DRAM row buffer locality, and it also reduces the access latency from two row buffer conflicts (one for the metadata and another for the datum itself). Based on this observation, they proposed to put the metadata for recently accessed rows in a small row buffer to the DRAM cache. Since metadata needed for data with temporal or spatial locality is cached on chip, it can be accessed with the similar latency as an SRAM tag store, while providing better energy efficiency than using a large SRAM tag store.

14.4.4 Parallel Hybrid Architecture

In the parallel hybrid architecture, both DRAM and PCM are used as the main memory. This architecture reduces the power budget because large portion of main memory is replaced by PCM.

In the hybrid memory, write-intensive accesses to PCM should be minimized due to its high write latency and lifetime limitation. Several works proposed to migrate write-intensive data/pages from PCM to DRAM to reduce the write accesses to PCM.

Dhiman et al. [11] proposed a hybrid main memory system that exposes both DRAM and PCM to software (operating system). If the number of writes to a particular PCM page exceeds a certain threshold, the contents of the page are copied to another page (either in DRAM or PCM), thus facilitating PCM wear leveling, while movement of hot pages to DRAM leads to saving of energy due to faster and more energy-efficient DRAM accesses.

Work in [43] suggested a management policy that monitors program access patterns and migrates hot pages to DRAM while keeping cold pages in PCM. In this way, the write-intensive operations are avoided, and thus the performance and energy saving are improved. Park et al. [39] proposed the main memory management mechanism of the operating system for the hybrid main memory. They proposed the migration scheme using access bits and power-off technique of DRAM chunk to mitigate background power consumption. Seok et al. [47] proposed a migration-based page caching technique for PCM-DRAM hybrid main memory system. Their technique aims to overcome the problem of the long latency and low endurance of PCM. For this, read-bound access pages are kept in PCM, and write-bound access pages are kept in DRAM. Their technique uses separate read and write queues for both PCM and DRAM and uses page monitoring to make migration decisions. Write-bound pages are migrated from PCM to DRAM, and read-bound pages are migrated from DRAM to PCM. The decision to migrate is taken as follows: when a write access is hit and the accessed page is in PCM write queue, it is migrated. Similarly, if a read access is hit and the accessed page is in the DRAM read queue, it is migrated.

Dynamically migrating data between DRAM and PCM can reduce the writes to PCM, but will bring energy cost and performance delay to the system. To handle the problem, reducing frequent page migration between PCM and DRAM is targeted at work [51]. Shin et al. proposed to store the page information and let it be visible to the operating system. With the information, page migration granularity can be dynamically changed based on whether the migration of pages is heavy. Heavy migration implies that pages which have similar access properties can be grouped to reduce the migration frequency.

Instead of dynamically migrating data between DRAM and PCM, a series of work [9, 16, 31] assumed that memory accessing patterns are given or can be predicted and proposed data allocation schemes in the hybrid memory to optimize different objectives. Choi et al. [9] developed the page-level allocation technique to

obtain the optimal performance, with well-designed proportion of DRAM's size to PCM's and proportion of DRAM's useful space to PCM's. Lee et al. [24] proposed a memory management algorithm which makes use of the write frequency as well as the recency of write references to accurately estimate future write references. The proposed scheme, which is applied on the parallel hybrid architecture, is compared to the scheme on DRAM-as-Cache architecture and is shown to be better in memory writes reduction. Liu et al. [31] developed variable-level partition schemes on the hybrid main memory to achieve the tradeoff between energy consumption and performance when the memory accesses and variables are given by the data-flow graph. Fu et al. [16] targeted at minimizing the energy consumption of the hybrid memory system while meeting the performance constraint and proposed a parallelism- and proximity-based variable partitioning scheme.

For task scheduling, Tian et al. [57] present a task-scheduling-based technique for addressing the challenges of hybrid DRAM/PCM main memory. They study the problem of task scheduling, assuming that a task should be entirely placed in either PCM bank or DRAM bank. Their approach works for different optimization objectives such as (1) minimizing the energy consumption of hybrid memory for a given PCM and DRAM size and given PCM endurance, (2) minimizing the number of writes to PCM for a given PCM and DRAM size and given threshold on energy consumption, and (3) minimizing PCM size for a given DRAM size, given threshold on energy consumption and PCM endurance.

14.5 Conclusion

In this chapter, emerging and nonvolatile memories and the state-of-the-art technologies in this area are introduced. A classification of different kinds of NVMs are firstly presented. Each NVM is introduced with a brief description to its features. Next, based on different objectives, several optimizations for memory architecture are introduced when NVM is working as on-chip cache and on-chip memory. Finally, we introduce that when NVM is working as the off-chip main memory, it is a widely used method to utilize both NVM and DRAM and combine them as a hybrid memory system. In the hybrid main memory, DRAM can be used either as a cache to the NVM main memory or as one of the memory partitions of the system. For both of the hybrid architecture, we present a series of schemes for performance and energy optimizations.

References

1. International Technology Roadmap for Semiconductors, 2007
2. https://www.semiconportal.com/en/archive/news/news-by-sin/130823-sin-panasonic-reram-production.html
3. http://loto.sourceforge.net/feram/doc/film.xhtml#(4)
4. http://www.alldatasheet.com/datasheet-pdf/pdf/465689/TI1/MSP430.html

5. Ahn J, Choi K (2012) Lower-bits cache for low power STT-RAM caches. In: International symposium on circuits and systems (ISCAS), pp 480–483
6. Chen Y, Wong WF, Li H, Koh CK, Zhang Y, Wen W (2013) On-chip caches built on multilevel spin-transfer torque RAM cells and its optimizations. J Emerg Technol Comput Syst 9(2):16:1–16:22. doi:10.1145/2463585.2463592
7. Chen YT, Cong J, Huang H, Liu B, Liu C, Potkonjak M, Reinman G (2012) Dynamically reconfigurable hybrid cache: an energy-efficient last-level cache design. In: Design, automation test in Europe conference exhibition (DATE), pp 45–50. doi:10.1109/DATE.2012.6176431
8. Cho S, Lee H (2009) Flip-n-write: a simple deterministic technique to improve PRAM write performance, energy and endurance. In: Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture, MICRO 42. ACM, pp 347–357
9. Choi JH, Kim SM, Kim C, Park KW, Park KH (2012) Opamp: evaluation framework for optimal page allocation of hybrid main memory architecture. In: Proceedings of the 2012 IEEE 18th international conference on parallel and distributed systems, ICPADS'12. IEEE Computer Society, pp 620–627
10. Dawber M, Rabe KM, Scott JF (2005) Physics of thin-film ferroelectric oxides. Rev Mod Phys 77:1083–1130. doi:10.1103/RevModPhys.77.1083
11. Dhiman G, Ayoub R, Rosing T (2009) PDRAM: a hybrid PRAM and DRAM main memory system. In: Proceedings of the 46th annual design automation conference, DAC'09. ACM, pp 664–469
12. Diao Z, Li Z, Wang S, Ding Y, Panchula A, Chen E, Wang LC, Huai Y (2007) Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory. J Phys 19(16):13
13. Dong X, Wu X, Sun G, Xie Y, Li H, Chen Y (2008) Circuit and microarchitecture evaluation of 3d stacking magnetic RAM (MRAM) as a universal memory replacement. In: Design automation conference (DAC), pp 554–559
14. Dong X, Xu C, Xie Y, Jouppi N (2012) Nvsim: a circuit-level performance, energy, and area model for emerging nonvolatile memory. IEEE Trans Comput-Aided Des Integr Circuits Syst (TCAD) 31(7):994–1007
15. Ferreira AP, Zhou M, Bock S, Childers B, Melhem R, Mossé D (2010) Increasing PCM main memory lifetime. In: Proceedings of the conference on design, automation and test in Europe, DATE'10. European Design and Automation Association, pp 914–919
16. Fu C, Zhao M, Xue CJ, Orailoglu A (2014) Sleep-aware variable partitioning for energy-efficient hybrid PRAM and DRAM main memory. In: Proceedings of the 2014 international symposium on low power electronics and design, ISLPED'14. ACM, pp 75–80
17. Guo X, Ipek E, Soyata T (2010) Resistive computation: avoiding the power wall with low-leakage, STT-MRAM based computing. In: International symposium on computer architecture (ISCA), pp 371–382
18. Inoue IH, Yasuda S, Akinaga H, Takagi H (2008) Nonpolar resistance switching of metal/binary-transition-metal oxides/metal sandwiches: homogeneous/inhomogeneous transition of current distribution. Phys Rev B 77:035,105. doi:10.1103/PhysRevB.77.035105
19. Jog A, Mishra AK, Xu C, Xie Y, Narayanan V, Iyer R, Das CR (2012) Cache revive: architecting volatile STT-RAM caches for enhanced performance in CMPs. In: Design automation conference (DAC), pp 243–252. doi:10.1145/2228360.2228406
20. Jung J, Nakata Y, Yoshimoto M, Kawaguchi H (2013) Energy-efficient spin-transfer torque RAM cache exploiting additional all-zero-data flags. In: International symposium on quality electronic design (ISQED), pp 216–222
21. Kim YB, Lee SR, Lee D, Lee CB, Chang M, Hur JH, Lee MJ, Park GS, Kim CJ, Chung Ui, Yoo IK, Kim K (2011) Bi-layered RRAM with unlimited endurance and extremely uniform switching. In: Symposium on VLSI technology (VLSIT), pp 52–53
22. Lee BC, Ipek E, Mutlu O, Burger D (2009) Architecting phase change memory as a scalable DRAM alternative. In: Proceedings of the 36th annual international symposium on computer architecture (ISCA), pp 2–13

23. Lee BC, Ipek E, Mutlu O, Burger D (2009) Architecting phase change memory as a scalable DRAM alternative. SIGARCH Comput Archit News 37(3):2–13
24. Lee S, Bahn H, Noh SH (2011) Characterizing memory write references for efficient management of hybrid PCM and DRAM memory. In: Proceedings of the 2011 IEEE 19th annual international symposium on modelling, analysis, and simulation of computer and telecommunication systems, MASCOTS'11. IEEE Computer Society, pp 168–175
25. Lee S, Bahn H, Noh SH (2014) Clock-dwf: a write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures. IEEE Trans Comput 63(9): 2187–2200
26. Li H, Chen Y (2009) An overview of non-volatile memory technology and the implication for tools and architectures. In: Design, automation test in Europe conference exhibition (DATE), pp 731–736
27. Li Q, Li J, Shi L, Xue CJ, He Y (2012) Mac: migration-aware compilation for STT-RAM based hybrid cache in embedded systems. In: International symposium on low power electronics and design (ISLPED), pp 351–356
28. Li Q, Li J, Shi L, Zhao M, Xue C, He Y (2014) Compiler-assisted STT-RAM-based hybrid cache for energy efficient embedded systems. IEEE Trans Very Large Scale Integr (VLSI) Syst 22(8):1829–1840
29. Li Q, Zhao M, Hu J, Liu Y, He Y, Xue CJ (2015) Compiler directed automatic stack trimming for efficient non-volatile processors. In: Annual design automation conference (DAC), pp 183:1–183:6
30. Li Y, Chen Y, Jones AK (2012) A software approach for combating asymmetries of non-volatile memories. In: International symposium on low power electronics and design (ISLPED), pp 191–196
31. Liu T, Zhao Y, Xue CJ, Li M (2011) Power-aware variable partitioning for dsps with hybrid PRAM and DRAM main memory. In: Proceedings of the 48th design automation conference, DAC'11. ACM, pp 405–410
32. Liu Y, Yang H, Wang Y, Wang C, Sheng X, Li S, Zhang D, Sun Y (2014) Ferroelectric nonvolatile processor design, optimization, and application. In: Xie Y (ed) Emerging memory technologies. Springer New York, pp 289–322. doi:10.1007/978-1-4419-9551-3_11
33. Meza J, Chang J, Yoon H, Mutlu O, Ranganathan P (2012) Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. IEEE Comput Archit Lett 11(2):61–64
34. Mittal S, Vetter J, Li D (2014) Lastingnvcache: a technique for improving the lifetime of non-volatile caches. In: IEEE computer society annual symposium on VLSI (ISVLSI), pp 534–540. doi:10.1109/ISVLSI.2014.69
35. Mittal S, Vetter J, Li D (2015) A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. IEEE Trans Parallel Distrib Syst 26(6):1524–1537
36. Mittal S, Vetter JS, Li D (2014) Writesmoothing: improving lifetime of non-volatile caches using intra-set wear-leveling. In: Proceedings of the 24th edition of the Great Lakes symposium on VLSI (GLSVLSI), pp 139–144
37. Papandreou N, Pozidis H, Pantazi A, Sebastian A, Breitwisch M, Lam C, Eleftheriou E (2011) Programming algorithms for multilevel phase-change memory. In: IEEE international symposium on circuits and systems (ISCAS), pp 329–332
38. Park H, Yoo S, Lee S (2011) Power management of hybrid DRAM/PRAM-based main memory. In: Proceedings of the 48th design automation conference, DAC'11. ACM, pp 59–64
39. Park Y, Shin DJ, Park SK, Park KH (2011) Power-aware memory management for hybrid main memory. In: 2011 The 2nd international conference on next generation information technology (ICNIT), pp 82–85
40. Quan B, Zhang T, Chen T, Wu J (2012) Prediction table based management policy for STT-RAM and SRAM hybrid cache. In: International conference on computing and convergence technology (ICCCT), pp 1092–1097

41. Qureshi MK, Karidis J, Franceschini M, Srinivasan V, Lastras L, Abali B (2009) Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In: Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture, MICRO 42. ACM, pp 14–23
42. Qureshi MK, Srinivasan V, Rivers JA (2009) Scalable high performance main memory system using phase-change memory technology. In: Proceedings of the 36th annual international symposium on computer architecture, ISCA'09. ACM, pp 24–33
43. Ramos LE, Gorbatov E, Bianchini R (2011) Page placement in hybrid memory systems. In: Proceedings of the international conference on supercomputing, ICS'11. ACM, pp 85–95
44. Raoux S, Burr G, Breitwisch M, Rettner C, Chen Y, Shelby R, Salinga M, Krebs D, Chen SH, Lung H, Lam C (2008) Phase-change random access memory: a scalable technology. IBM J Res Dev 52(4.5):465–479. doi:10.1147/rd.524.0465
45. Rasquinha M, Choudhary D, Chatterjee S, Mukhopadhyay S, Yalamanchili S (2010) An energy efficient cache design using spin torque transfer (STT) RAM. In: International symposium on low power electronics and design (ISLPED), pp 389–394
46. Rogers BM, Krishna A, Bell GB, Vu K, Jiang X, Solihin Y (2009) Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In: International symposium on computer architecture (ISCA), pp 371–382
47. Seok H, Park Y, Park KH (2011) Migration based page caching algorithm for a hybrid main memory of DRAM and PRAM. In: Proceedings of the 2011 ACM symposium on applied computing, SAC'11. ACM, pp 595–599
48. Seong NH, Woo DH, Lee HHS (2010) Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. SIGARCH Comput Archit News 38(3):383–394
49. Sharifi A, Kandemir M (2011) Automatic feedback control of shared hybrid caches in 3D chip multiprocessors. In: International conference on parallel, distributed and network-based processing (PDP), pp 393–400
50. Sharifi A, Kandemir M (2013) Using cache-coloring to mitigate inter-set write variation in non-volatile caches. In: Iowa State University, Ames, Technical report
51. Shin DJ, Park SK, Kim SM, Park KH (2012) Adaptive page grouping for energy efficiency in hybrid PRAM-DRAM main memory. In: Proceedings of the 2012 ACM research in applied computation symposium, RACS'12. ACM, pp 395–402
52. Smullen C, Mohan V, Nigam A, Gurumurthi S, Stan M (2011) Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In: International symposium on high performance computer architecture (HPCA), pp 50–61
53. Sun G, Dong X, Xie Y, Li J, Chen Y (2009) A novel architecture of the 3D stacked MRAM l2 cache for CMPS. In: International symposium on high performance computer architecture (HPCA), pp 239–249
54. Sun G, Kursun E, Rivers JA, Xie Y (2013) Exploring the vulnerability of CMPS to soft errors with 3D stacked nonvolatile memory. J Emerg Technol Comput Syst 9(3):22:1–22:22. doi:10.1145/2491679
55. Sun Z, Bi X, Li HH, Wong WF, Ong ZL, Zhu X, Wu W (2011) Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In: International symposium on microarchitecture (MICRO), pp 329–338
56. Sun Z, Wu W, Li H (2013) Cross-layer racetrack memory design for ultra high density and low power consumption. In: Design automation conference (DAC), pp 1–6
57. Tian W, Zhao Y, Shi L, Li Q, Li J, Xue CJ, Li M, Chen E (2013) Task allocation on nonvolatile-memory-based hybrid main memory. IEEE Trans Very Large Scale Integr Syst 21(7): 1271–1284
58. Venkatesan R, Kozhikkottu V, Augustine C, Raychowdhury A, Roy K, Raghunathan A (2012) Tapecache: a high density, energy efficient cache based on domain wall memory. In: International symposium on low power electronics and design (ISLPED), pp 185–190
59. Venkatesan R, Kozhikkottu V, Augustine C, Raychowdhury A, Roy K, Raghunathan A (2012) Tapecache: a high density, energy efficient cache based on domain wall memory. In: International symposium on low power electronics and design (ISLPED), pp 185–190

60. Venkatesan R, Sharad M, Roy K, Raghunathan A (2013) DWM-tapestri – an energy efficient all-spin cache using domain wall shift based writes. In: Design, automation & test in Europe conference & exhibition (DATE), pp 1825–1830
61. Wang J, Dong X, Xie Y, Jouppi N (2013) i2wap: improving non-volatile cache lifetime by reducing inter- and intra-set write variations. In: International symposium on high performance computer architecture (HPCA2013), pp 234–245. doi:10.1109/HPCA.2013.6522322
62. Wang Y, Liu Y, Li S, Zhang D, Zhao B, Chiang MF, Yan Y, Sai B, Yang H (2012) A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In: Proceedings of the ESSCIRC (ESSCIRC), pp 149–152
63. Wu X, Li J, Zhang L, Speight E, Rajamony R, Xie Y (2009) Hybrid cache architecture with disparate memory technologies. In: Proceedings of the 36th annual international symposium on computer architecture (ISCA), pp 34–45
64. Xu W, Sun H, Wang X, Chen Y, Zhang T (2011) Design of last-level on-chip cache using spin-torque transfer RAM (STT RAM). IEEE Trans Very Large Scale Integr (VLSI) Syst 19(3):483–493
65. Xue CJ, Zhang Y, Chen Y, Sun G, Yang JJ, Li H (2011) Emerging non-volatile memories: opportunities and challenges. In: Proceedings of international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 325–334
66. Yang BD, Lee JE, Kim JS, Cho J, Lee SY, gon Yu B (2007) A low power phase-change random access memory using a data-comparison write scheme. In: IEEE international symposium on circuits and systems, ISCAS'07, pp 3014–3017
67. Yazdanshenas S, Pirbasti M, Fazeli M, Patooghy A (2014) Coding last level STT-RAM cache for high endurance and low power. Comput Archit Lett 13(2):73–76
68. Yoon H (2012) Row buffer locality aware caching policies for hybrid memories. In: Proceedings of the 2012 IEEE 30th international conference on computer design, ICCD'12. IEEE Computer Society, pp 337–344
69. Yoon H, Meza J, Harding R, Ausavarungnirun R, Mutlu O (2011) Dynrbla: a high-performance and energy-efficient row buffer locality-aware caching policy for hybrid memories. SAFARI Technical Report No. 2011–005
70. Yun J, Lee S, Yoo S (2012) Bloom filter-based dynamic wear leveling for phase-change RAM. In: Proceedings of the conference on design, automation and test in Europe, DATE'12. EDA Consortium, pp 1513–1518
71. Zhang W, Li T (2009) Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In: Proceedings of the 2009 18th international conference on parallel architectures and compilation techniques, PACT'09. IEEE Computer Society, pp 101–112
72. Zhao M, Jiang L, Shi L, Zhang Y, Xue C (2015) Wear relief for high-density phase change memory through cell morphing considering process variation. IEEE Trans Comput-Aided Des Integr Circuits Syst 34(2):227–237
73. Zhao M, Li Q, Xie M, Liu Y, Hu J, Xue CJ (2015) Software assisted non-volatile register reduction for energy harvesting based cyber-physical system. In: Design, automation & test in Europe conference & exhibition (DATE), pp 567–572
74. Zhao W, Belhaire E, Mistral Q, Chappert C, Javerliac V, Dieny B, Nicolle E (2006) Macromodel of spin-transfer torque based magnetic tunnel junction device for hybrid magnetic-cmos design. In: IEEE international behavioral modeling and simulation workshop, pp 40–43
75. Zhou P, Zhao B, Yang J, Zhang Y (2009) A durable and energy efficient main memory using phase change memory technology. SIGARCH Comput Archit News 37(3):14–23
76. Zhou P, Zhao B, Yang J, Zhang Y (2009) Energy reduction for STT-RAM using early write termination. In: International conference on computer-aided design (ICCAD), pp 264–268
77. Zhu JG (2008) Magnetoresistive random access memory: the path to competitiveness and scalability. Proc IEEE 96(11):1786–1798. doi:10.1109/JPROC.2008.2004313
78. Zwerg M, Baumann A, Kuhn R, Arnold M, Nerlich R, Herzog M, Ledwa R, Sichert C, Rzehak V, Thanigai P, Eversmann BO (2011) An 82μA/MHz microcontroller with embedded feram for energy-harvesting applications. In: International solid-state circuits conference (ISSCC), pp 334–336

Network-on-Chip Design

15

Haseeb Bokhari and Sri Parameswaran

Abstract

Continuous transistor scaling has enabled computer architecture to integrate increasing numbers of cores on a chip. As the number of cores on a chip and application complexity has increased, the on-chip communication bandwidth requirement increased as well. Packet-switched network on chip (NoC) is envisioned as a scalable and cost-effective communication fabric for multi-core architectures with tens and hundreds of cores. In this chapter we focus on on-chip communication architecture design and introduce the reader to some essential concepts of NoC architecture. This is followed by a discussion on the commonly used power-saving techniques used for NoCs and the drawbacks and limitations of these techniques. We then concentrate on performance optimization through intelligent mapping of applications on multi-core architectures. We conclude the chapter with a discussion of various application-specific on-chip interconnect design methods.

Acronyms

CMOS	Complementary Metal-Oxide-Semiconductor
CMP	Chip Multi-Processor
DOR	Dimension Ordered Routing
DRAM	Dynamic Random-Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
IP	Intellectual Property
MPSoC	Multi-Processor System-on-Chip
NI	Network Interface
NoC	Network-on-Chip
SoC	System-on-Chip
TDMA	Time-Division Multiple Access

H. Bokhari (✉) • S. Parameswaran
University of New South Wales (UNSW), Sydney, NSW, Australia
e-mail: hbokhari@cse.unsw.edu.au; sridevan@cse.unsw.edu.au

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_16

Contents

15.1 On-Chip Interconnect Architecture

Every multi-core chip has two major on-chip components: processing elements (*core*) and other non-processing elements such as communication and memory architecture (*uncore*) [27]. Although high transistor density enables computer architects to integrate tens to hundreds of cores in a chip, the main challenge is to enable efficient communication between such a large number of on-chip components. The on-chip communication architecture is responsible for all memory transactions and I/O traffic and provides a dependable medium for inter-processor data sharing. The performance of on-chip communication plays a pivotal role in the overall performance of the multi-core architecture. The advantage of having multiple high-performance on-chip processors can easily be overturned by an underperforming on-chip communication medium. Hence, providing a scalable and high-performance on-chip communication is a key research area for multi-core architecture designers [17]. The main challenges faced by interconnect designers are:

- *Scalable communication for tens of cores*: It is fair to state that the performance of processing elements in multi-core chips can be communication constrained [17]. Due to ever-increasing improvement in processing capabilities, it is quite possible to have a wide gap between the data communication rate and the data consumption rate. With tens of on-chip components, it is not possible to have single-cycle communication latency between components placed at the far ends of a chip. Furthermore, with a large number of on-chip components, the on-chip interconnect is expected to support multiple parallel communication streams.
- *Limited power budget*: In 1974, Dennard predicted that the power density of transistor will remain constant as we move into lower node sizes. This is known as *Dennard's scaling law* [37]. However, in the last decade or so, researchers

have observed that the transistor's power cannot be reduced at the same rate as the area. Therefore, we are facing a situation where we have an abundance of on-chip transistors but do not have enough power to switch all these transistor at the same time, due to power and thermal constraints. Therefore, increasing the power efficiency of all on-chip components has become the main prerequisite to continue Moore's scaling. The on-chip communication architecture can consume roughly 19% of total chip power in a modern multi-core chip [35]. Therefore, it is a challenging task to design a power-efficient on-chip interconnect that can still satisfy the latency and bandwidth requirements of current and future applications.

- *Heterogeneous applications*: A modern multi-core chip is expected to execute a large set of diverse applications. Each application can interact with computing architecture in a unique way; hence, the communication latency and bandwidth requirement can vary across different applications [32]. For example, an application with a large memory footprint is expected to regularly generate cache misses and can hence be classified as a communication-bound application. The performance of such applications is highly correlated with the efficiency of interconnect. On the other hand, an application with a smaller memory footprint is expected to be processor bound and agnostic to on-chip interconnect properties. Therefore, on-chip interconnects are often designed for worst-case scenarios (memory-bound applications in this case) and can therefore be inefficient for processor-bound applications. The situation is aggravated when both memory- and processor-bound applications are executed at the same time.
- *Selecting interconnect performance metrics*: A major shortcoming in previous research is classifying the on-chip interconnect performance in terms of application-agnostic metrics such as transaction latency and memory bandwidth, instead of application-level performance metrics such as execution time and throughout [31, 70]. Therefore a major challenge is extracting the correct metric to evaluate different possible interconnect architecture design points for a given set of applications.
- *Chip design cost*: The cost of designing a multi-core chip has been increasing alarmingly due to high NRE cost (NRE Cost: *Nonrecurring engineering cost*. The term is used to classify the cost associated with researching, prototyping, and testing a new product.) associated with small node sizes. A major portion of total chip cost is reserved for design verification and testing. Therefore, designers are expected to reuse previously designed and verified on-chip interconnects for new chip designs to reduce cost. With a multitude of interconnect architectures available, it is important to incorporate time-efficient design space exploration tools in research phase to select the most suitable interconnect for a given set of target applications.
- *Interconnect reliability*: With reducing node size, the concerns about the reliability of the digital circuits are on the rise. Any unexpected change in operating conditions such as supply voltage fluctuation, temperature spikes, or a random alpha particle collision can cause erratic behavior in the output of a circuit. A soft error in on-chip interconnect can result in erroneous application output or system deadlock if the control data is corrupted. Multi-core systems are finding

their ways into reliability-critical applications such as autonomous driving cars and medical equipment. Therefore designers are expected to integrate varying levels of reliability features in on-chip interconnect under given power and area constraints.

- *Codesign of memory hierarchy and on-chip interconnect*: In modern multi-core architectures, on-chip memory hierarchy is closely coupled with on-chip interconnect architecture. In fact, for shared memory architectures, on-chip communication is the major factor in deciding the performance of memory hierarchy (cache, Dynamic Random-Access Memory (DRAM) controllers, etc.). Therefore interconnect designers are often faced with the challenge of exploring the combined design space of memory hierarchy and on-chip interconnect.

15.1.1 Bus-Based SoC Architectures

Traditionally, system-on-chip (SoC) designs used a very simple on-chip interconnect such as ad hoc point-to-point connections or buses. The bus-based architecture is perhaps the oldest on-chip interconnect standard in the computer industry and is still used in many System-on-Chip (SoC) applications [87]. The simplicity of protocol and hence low gate cost are possibly the main reasons that bus-based architectures have dominated all other available on-chip interconnect options. For a small number of on-chip components, bus interconnect is easier to integrate due to simple protocol design and is efficient in terms of both power and silicon cost. In bus-based architectures, multiple components interact using a single data and control bus, hence providing a simple master-slave connection. Arbitration is required when multiple masters try to communicate with a single slave, giving rise to resource contention. Hence the scalability of bus-based architecture in terms of performance is questionable in large SoC-based designs [61]. Some classic design techniques for bus-based SoCs proposed in [34, 41] use worst-case bus traffic to design optimal architecture. Kumar et al. [57] have given a detailed study of the scalability and performance of shared bus-based Chip Multi-Processor (CMP) architectures. They concluded that a bus-based interconnect network can significantly affect the performance of cache-coherent CMPs.

Several improvements to traditional bus-based interconnect architectures have been proposed. ARM Ltd., AMBA Architecture [1], IBM CoreConnect Architecture [3], and Tensilica PIF Interface [5] are few examples of the widely used advanced bus-based communication medium. All of these architectures provide several advanced functionalities like burst-based data transfers, multi-master arbitration, multiple outstanding transactions, bus locking, and simultaneous asynchronous and synchronous communication. However, Rashid et al. [91] have analytically showed that even advanced bus-based architectures like AMBA are outperformed by modern Network-on-Chip (NoC)-based communication architectures in terms of performance. However, the same study shows that designers are still inclined toward AMBA-based on-chip interconnects due to the area and energy overhead of modern NoC designs. SoC designers have a keen interest in fair comparison of various

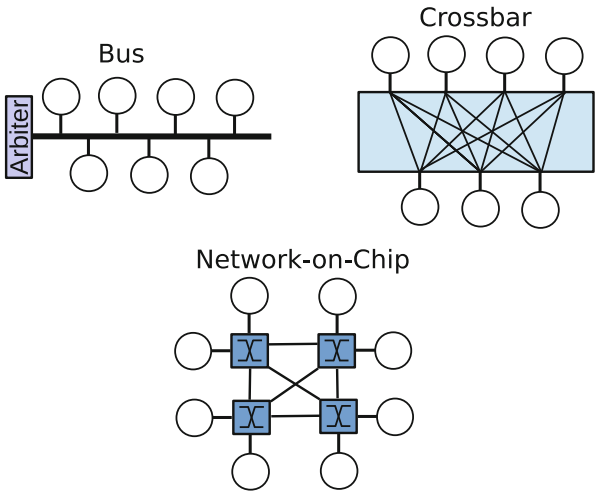


Fig. 15.1 Evolution of on-chip interconnect

commercial bus architectures; however, the performance of on-chip interconnects greatly depends on each application's traffic pattern, bus microarchitecture, and system parameters [67].

The simplicity of the bus-based architecture design, predictable access latency, and low area overhead are the key selling points. However, beyond a small number of cores, the performance of the bus interconnect degrades significantly [83].

15.1.2 Crossbar-on-Chip Interconnect

Single shared bus architecture is evidently slower in the case of multiple master-slave data transactions. The prime bottleneck is the single shared medium and latency due to arbitration among many master interfaces (Fig. 15.1). Therefore, the first approach to design a scalable on-chip interconnect was adaption of crossbar topology. A crossbar is a matrix switch fabric that connects all the inputs with all the outputs enabling multiple communication connections (Fig. 15.1). The idea has been borrowed from the telecommunication industry where such architectures have been successfully used for four decades in telephony applications [82].

The same concept of multiple communication channels was implemented in the SoC design industry by combining multiple shared buses to form an all-input-all-output connection matrix. The concept is also known as hierarchal bus or multilayer bus. STBus [53] is perhaps the best-known commercial bus architecture that inherently supports crossbar architectures. A design methodology for AMBA-based cascaded bus architecture is provided by Yoo [104]. Yoo et al. have experimented with integrating 90 IP blocks in a single crossbar-based SoC design. Similarly, authors in [72] have provided a complete design methodology for designing an

application-specific crossbar architecture using the STBus protocol. They claim significant performance improvements over standard single-bus architectures. The most interesting crossbar implementation is the interconnect system for IBM Cyclops64 architecture. Each Cyclops64 crossbar connects 80 custom processors and about 160 memory banks. With single transaction latency of seven cycles and bandwidth comparable to state-of-the-art NoC architecture, Cyclops64 interconnects is perhaps the most advanced practical crossbar design for the SoC domain.

Researchers have been arguing over the scalability of crossbar-based interconnect architecture due to the nonlinear relation between the number of the ports and latency and wire cost [107]. However, recent experiments from [82] show that a 128×128 port crossbar in a 90 nm technology is feasible. They have benchmarked their crossbar design against state-of-the-art mesh-based NoC design and concluded that the crossbar design matched NoC architectures in terms of latency, bandwidth, and power consumption. However, the design complexity is prohibitively high due to complex wire layouts.

15.1.3 Network-on-Chip Interconnect

Following Moore's law of available on-chip transistor resources, we are looking beyond having thousands of cores on a single chip. It has been predicted that the performance of such kilo-core Multi-Processor System-on-Chip (MPSoC) will be communication architecture dependent [16]. Traditional bus-based architectures cannot scale beyond a few tens of IP blocks, and there is a need to provide a more scalable and protocol invariant communication architecture.

The solution to the scalability problem of bus-based architectures was found in the form of network-on-chip architectures [29, 58]. NoC inherently supports the general trend of highly integrated SoC design and provides a new de facto standard of on-chip communication design. The basic idea of NoC has been adapted from the well established structure of computer networks. In particular, the layered service architecture of computer networks has been well adapted in NoC to provide a scalable solution. In a NoC architecture, data is converted into packets, and these packets traverse number of hops (switches or routers) based on a predefined routing technique. The key advantages of using NoC as the on-chip interconnect are:

- NoCs inherently support multiple communication paths through a combination of physically distributed routers and links, which greatly increases the available on-chip data bandwidth. This enables different cores to exchange data in parallel without any central arbitration. This makes NoC an ideal candidate interconnect for supporting increasing communication needs of multi-core chips with tens and hundreds of cores. Multiple communication paths between given source and destination cores give NoC an inherent fault tolerance. In case of permanent error in the router or link on a given path, data can be rerouted through an alternate path between source and destination cores.

- NoC architectures use short electric wires that have highly predictable electric properties. Compared to bus interconnect, smaller drive transistors are required to switch short wires between routers. This helps to improve the *energy/bit* metric of interconnects. Moreover, due to shorter wire delays, NoCs can be switched at higher frequencies than buses and crossbars without any significant increase in power. Deep sub-micron semiconductor manufacturing introduces integrity issues in wires. Having shorter wires reduces the probability of manufacturing faults and hence improve the production yield. The predictable electric properties of short wires also help in reducing design verification cost.
- NoCs follow a modular design paradigm by allowing reuse of existing hardware Intellectual Property (IP) blocks. For most designs, NoCs can be easily scaled for different number of cores and applications by simply instantiating multiple copies of existing designed and verified router IPs. This reduces the overall complexity of the chip design process.
- NoCs provide a clear boundary between computation and communication. On-chip components (memory controllers, processing cores, hardware IPs, etc.) can have different communication protocols (AXI, AHB, etc.) which are converted to a standard packet format through the help of protocol convertors. Therefore, data communication between on-chip components is agnostic of communication protocol used by different components. This is a useful feature for designing heterogeneous SoCs with hardware components selected from different IP vendors.

15.2 Defining Features of Network on Chip

NoC-based MPSoC designs have attracted attention of researchers for the last decade. The defining features of NoC design are router design, routing algorithms, buffer sizing, flow control, and network topology. We will discuss them in more detail.

15.2.1 Topology

A NoC consists of routers and communication channels. NoC topology defines the physical layout of the routers and how these routers are connected with each other using the communication channels. The selection of NoC topology can have a significant effect on multi-core performance, power budget, reliability, and overall design complexity. For example, if the average number of hops between nodes is high, packets have to travel longer and hence network latency will be high. Similarly, if a topology requires very long physical links, designers have to make an effort to ensure timing closure for longer links. Moreover, topologies that allow diverse paths between nodes can potentially provide higher bandwidth and also prove to be more

reliable in the case of faulty links. Therefore, when designing NoC-based multi-core systems, the first decision is to choose the NoC topology [83].

NoC topologies can be classified as *direct* and *indirect* topologies [83]. In direct topologies, each on-chip component such as core or memory, is connected with a router, and therefore each router is used to inject or eject traffic from the network. In indirect topologies, the on-chip components are connected only to *terminal* routers, whereas the other routers only forward the traffic [83]. The *degree* parameter of a router defines the number of neighboring routers and on-chip component to which the router has links. The degree parameter defines the number of input/output ports in each router. Note that the complexity of router microarchitecture increases with an increase in the degree of router.

Figure 15.2 shows three commonly used direct topologies *ring*, *mesh*, and *torus*. The ring is the simplest topology to implement in terms of silicon area and design complexity. The degree of each router in ring interconnect is 3 (two neighbor router + one local resource (core, memory, etc.)). The drawback of ring topology is performance scalability. The number of hops between two nodes in worst-case scenarios is proportional to N: the number of nodes in the topology. Furthermore, rings provide limited bandwidth and are less reliable due to poor path diversity. Therefore, rings become impractical for multi-core chips with more than 8–16 nodes [8].

Mesh and torus topologies solve the scalability problems of ring topology, albeit at a cost of higher degree routers and possibly more complex VLSI layout. Each router in a mesh topology has a degree of 5, except for the routers on the border. Torus can be classified as an enhanced form of mesh with wrap around links between border routers. These links use router ports that are not required to implement mesh topology. The wraparound links in torus reduce the average number of hops and

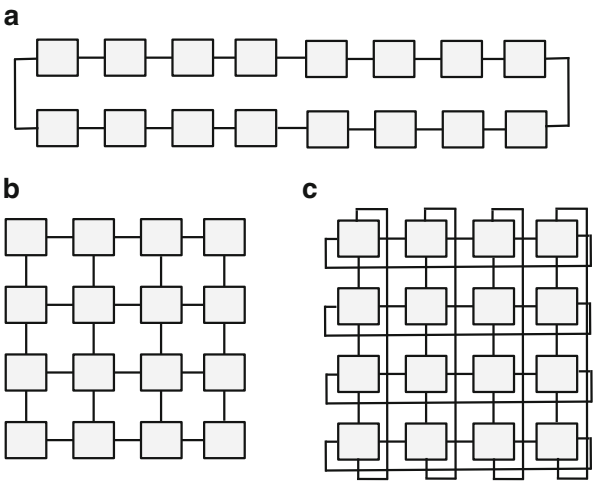


Fig. 15.2 Well known direct topologies. (**a**) Ring. (**b**) Mesh. (**c**) Torus

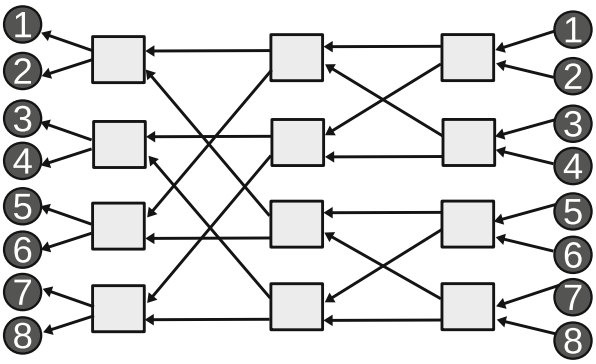


Fig. 15.3 2-ary 3-fly Butterfly topology

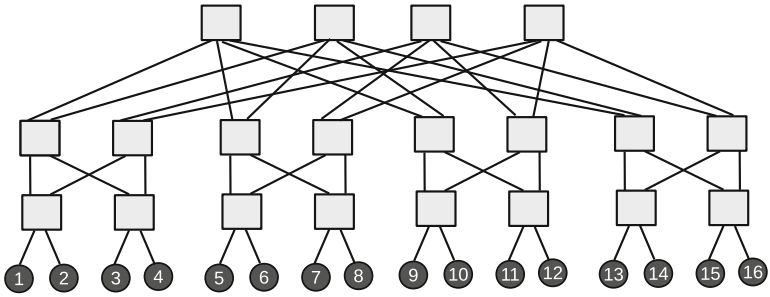


Fig. 15.4 3 Tier Fat Tree topology

provide better bisection bandwidth. The worst-case number of hops is $\sqrt{N}+1$ for torus and $2\sqrt{N}-1$ for mesh. In mesh, all communication links are short and equal in size. However, for torus wraparound links are considerably longer and need special attention for timing closure.

Two classical examples of indirect networks, *butterfly* and *fat tree* are shown in Figs. 15.3 and 15.4, respectively. The important feature of butterfly topology is that the hop distance between any source-destination node pair is fixed (three in the topology shown in Fig. 15.3). The router has degree 2 (two input and two output ports), resulting in low-cost routers. However, the number of routers is greater than the number of SoC components. The two main disadvantages of butterfly topologies are single communication path between a given source-destination pair resulting in low bandwidth and low link fault tolerance and more complex wire layout due to uneven link lengths. The fat tree topology provides higher bandwidth and excellent diversity of possible routing paths. However, these qualities come at a cost of silicon area (more routers) and complex wire layout.

In addition to these regular topologies, application-specific MPSoCs are often designed on top of NoCs with customized topologies [20]. The data traffic patterns

are often known at design time, and therefore a communication graph can be extracted from the application specifications [12, 89]. The communication graph combined with knowledge of the physical mapping of SoC components can be used to create a topology that meets certain performance, energy, or area constraints [13, 20, 95].

15.2.2 Routing

The routing algorithm defines the sequence of routers between source and destination nodes that a data packet will traverse. The quality of the routing algorithm is determined by the average packet latency and power consumption. A good routing algorithm evenly distributes the traffic across all the routers and maximizes the saturation throughput of the network. Power can be optimized by keeping the routing circuit simple and keeping the number of hops traveled by data packets low [83].

Deterministic routing is the simplest routing scheme and is widely used in NoCs. For a given source and destination pair, data packets always travel through a predefined set of routers. Dimension Ordered Routing (DOR) for mesh is a common example of deterministic routing. In XY routing for mesh, depending on the physical location of the source and destination pair, the packet always travels first in the X (horizontal) direction and then in the Y (vertical) direction. However, deterministic routing can cause traffic hotspots in the case of an asymmetrical communication pattern between nodes [63, 83].

Oblivious routing is superior to deterministic routing in terms of path selection. For a given source-destination pair, oblivious routing can select one of many possible routes. However, this decision is taken without any knowledge of the network's current traffic condition. One example for oblivious routing is ROMM [76]. In the ROMM routing scheme for mesh, an intermediate node is selected at random on minimal paths between source and destination, and the data packet is first sent to the intermediate node and from there to the destination using a deterministic routing algorithm.

Adaptive routing is the most robust routing scheme, as it uses global knowledge about the current network traffic state to select the optimal path [63]. Adaptive routing distributes the traffic node across different network routers and hence maximally utilize the network bandwidth. However, implementing adaptive routing increases the design complexity of the routers [83]. Moreover, there is always a limitation on how much global knowledge can be forwarded to each router, hence limiting the effectiveness of the routing scheme [63].

In addition to standard routing schemes, MPSoC designers often use application-specific routing schemes for NoCs [22, 28, 79]. Application communication graphs can be analyzed to extract information about data volume and criticality. This information can be used to design routing algorithms that minimize the communication latency [79].

15.2.3 Flow Control

Flow control determines how data is transferred between different routers in a NoC. Specifically, flow control dictates the buffer and link allocation schemes. The design objective for flow control architecture is to minimize the buffer size and hence silicon area and power of routers and to keep the network latency low. In packet-switched NoCs, a data message is broken into a predefined *packet* format. A network packet size can be further broken and serialized into multiple *flits*. The size of flit normally equals physical channel width [83]. Additional information is added to each flit to indicate *header*, *body*, and tail flit. The routing and other control information can either be added only to the header flit or it can be added to each flit depending on implementation.

In *store-and-forward* flow control [30], before forwarding the packet to the next node, the router waits until the whole packet has been transmitted into its local buffer. This means that the input buffer must have enough space to store the whole packet, which can increase router area and power consumption. This scheme also increases the communication latency, as packets spend a long time at each node just waiting for buffering, although the output port might be free.

Virtual cut-through [54] improves on store-and-forward flow control by allowing a packet to be routed to the next router even before the whole packet arrives at the current router. However, the packet is only forwarded if the downstream router has enough buffer space to store the complete packet. This means that buffer size remains the same as in the case of store-and-forward flow control with improvement in per-hop latency.

Wormhole routing [90] is a more robust scheme, as it allocates buffer space at the granularity of flit, opposed to the virtual cut-through and store-and-forward scheme which allocates buffers at the granularity of packet. As soon as flit of a packet arrives at an input port, it can be forwarded even if only one flit space is available in the input port of the next router (and output channel is not allocated). The wormhole flow control scheme results in low-area routers, and it is therefore widely used in most on-chip networks [83]. The term *wormhole* implies that a single packet can span multiple routers at the same time. The main downside of this scheme is that the multiple links can be blocked at the same time in case the header flit of a multiple flit packet is blocked in one of the routers on the communication path.

15.2.4 Router Microarchitecture

The key building block of NoC is the router. The router's microarchitecture dictates the silicon area, the power, and, most importantly, the performance of the NoC. The maximum frequency at which a router can operate depends on the complexity of the logic used in the router microarchitecture, which in turn translates into higher-level performance metrics such as network latency and maximum achievable bandwidth. The complexity of the router's microarchitecture depends on the network topology (degree), flow control method, and routing algorithms. For

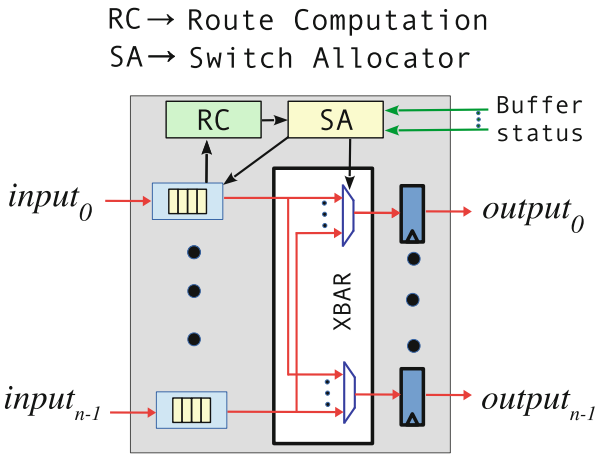


Fig. 15.5 Wormhole router architecture

example, a complex adaptive routing algorithm can be used to improve the worst-case network bandwidth, but it will result in increased area and power.

Figure 15.5 shows the overall architecture of a packet switch wormhole router [38]. The basic building blocks of a wormhole router are *input buffer*, *route computation*, *switch allocators*, and *crossbar switch*. Input buffers store flits when they arrive in the router and keep them stored until they are forwarded to the next router. The route computation unit calculates the output port for the header flit stored at the head of each input buffer, depending on the routing algorithm. The switch allocator arbitrates between different packets contending for the same output ports. The crossbar switch is logically a set of muxes that route flits from the input port to the output port. The data from the crossbar is stored in output buffers which can be as simple as flip-flops. The input buffers also propagate the buffer occupancy status to neighboring routers to implement flow control [38].

15.2.4.1 Progress of a Packet in a Wormhole Router

The incoming flit is first stored in the input buffer (Buffer Write (*BW*) stage). If the flit at the front of the input buffer is the header flit, the route computation unit calculates the output port required to send the packet to its destination and asserts the port request signal to the switch allocator (route computation (*RC*) stage). In modern routers, the switch allocator consists of multiple port arbiters, one for each output port. Each output arbiter chooses one of the multiple input requests for a given output port. When an output port is allocated to an input port, the specific output port is locked until the whole packet (tail flit) has crossed the crossbar switch. Before doing anything further, the switch allocator checks if the downstream router has enough space to store the outgoing flit. If the buffers are full at the downstream routers, the switch allocator blocks the packet transmission. However, if buffer space is available, switch allocator sets the proper select lines for the crossbar switch and

also instructs the input buffer to remove the flit at the front. This whole process is called the switch allocation (*SA*) stage. On getting a valid signal and output port selection from the switch allocator, the crossbar switch transfers the flit from the input port to the output port (switch traversal *ST* stage). The flit from the output port of the router then travels over wire links to get latched in the input buffer of the downstream router (link traversal (*LT*) stage). Note that the header flit of a packet goes through all stages discussed here. The body and tail flit, however, skip the RC and SA stages, as the output had already been reserved by the header flit.

15.2.4.2 Optimization and Logic Synthesis of Routers

Executing all router stages in a single cycle can be achieved at a lower frequency because the cumulative logic delay of stages can be long. Single-cycle operation might require a higher supply voltage depending on the target frequency which can increase the power consumption. Therefore, most of the high-performance routers are often pipelined [30, 38]. However, increasing the number of pipeline stages increases the per-hop latency and hence the overall network latency. The number of pipeline stages also depends on the sophistication of the RC, SA, and ST stages.

In commonly used pipelined routers, the LT and BW are done in one cycle, and RC, SA, and ST are executed in the next cycle. However in the case of more complex router architectures, the second pipeline stage can be further divided into RC+SA and ST pipeline stages. The pipeline stages can affect the area and power consumption [11]. Using fewer pipeline stages results in more stringent latency constraints for logic synthesis, and hence the synthesis tool has to insert larger gates with lower delays. Larger logic gates have higher dynamic and leakage power. Pipeline stages on the other hand can reduce the gate sizes; however, the overall area may increase due to addition of the pipeline flip-flops [11]. Therefore, the logic synthesis of routers is a classic power-performance-area tradeoff problem [11, 38, 81].

15.2.5 Network Interface

Router is the main building block of NoC and carries the burden of routing the packets across network. *Network Interface (NI)* on the other hand acts as a *bridge* between hardware IP blocks and NoC. Figure 15.6 shows an example of NI signaling scheme. NI converts IP block's communication protocol to NoC's packet format and performs associated housekeeping operations. The communication protocol can vary across IP vendors. For example, most of the ARM IPs normally support AXI protocol [6], whereas Xtensa processors use PIF protocol [5]. Therefore, NIs actually enable the modular property of NoC by letting different IP communicate seamlessly irrespective of their communication protocol. An example MPSoC that contains IPs from different vendors is shown in Fig. 15.7. To enable maximum flexibility to system designers, commercial NoC vendors provide support for multiple communication protocols. For example, Sonics [4] supports AXI and OCP protocols, and Arteris [7] supports AMBA, PIF, BVCI, and OCP protocols.

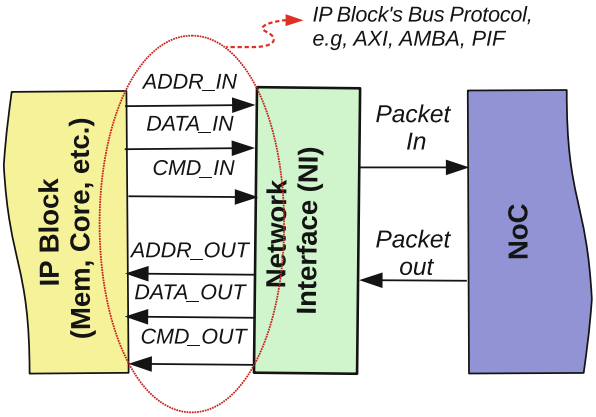


Fig. 15.6 Network interface (NI)

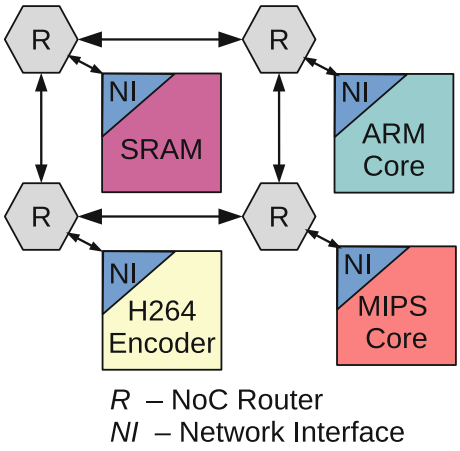


Fig. 15.7 Heterogeneous MPSoC with IPs from different vendors

Microarchitecture of the NI depends on the programming model selected for SoC [36]. For example, Tilera iMesh on-chip interconnect network [103] supports both MPI [62] and shared memory [105] programming models. iMesh enables parallel operation of these programming models by integrating separate NIs and NoCs for both of these programming models.

Researchers have proposed integrating various advance features in NI design with primary focus on quality-of-service (QoS) services. Radulescu et al. [85] proposed a NI design that integrates QoS services for shared memory MPSoCs, with support for both guaranteed and best-effort services. Mishra et al. [70] proposed keeping track of vital application information such as cache miss rate and executed instructions in NI to support fair QoS among heterogeneous applications. Similarly Chen et al. [25] monitor application cache miss rate and other processing core's information in NI logic to implement NoC power optimization.

15.2.6 Performance Metrics

As described earlier in chapter, there are number of possible customizable features in a NoC. Therefore, it is important to discuss metrics that are often used to assess the NoC's performance.

NoC performance is often evaluated using network *latency* and *throughput* [30]. An example latency versus traffic injection rate is shown in Fig. 15.8. The *latency* is defined as average time it takes for packets to travel between source-destination pairs in NoC. Network latency can calculated as sum of latency experienced at each hop (router). The *zero-load latency* metric defines the lower bound on latency when there are no other traffic in the network. However, as traffic is introduced in the NoC at a higher rate, packets travel slower in the network due to channel contention. The *saturation throughput* point is defined as injection rate at which packet latency becomes prohibitively large. Some research also define *saturation throughput* as the injection rate at which the average network latency is roughly three times the zero-load latency [50]. As a rule of thumb, designers aim to minimize the zero-load latency and maximize the saturation throughput. In the absence of real applications, *latency* and *throughput* are commonly used to evaluate different NoC architecture using different synthetic traffic patterns [64].

Although latency and throughput are easier to evaluate, researchers argue that these metrics may be misleading for assessing impact of NoC on overall system performance. Mishra et al. [70] showed that some applications are network latency sensitive, whereas some applications are bandwidth sensitive. Therefore, using application-level metrics such as execution time and average memory access latency is a better way to evaluate NoC performance. Similarly Chen et al. [25] showed that the impact of NoC latency on application's performance depends on other

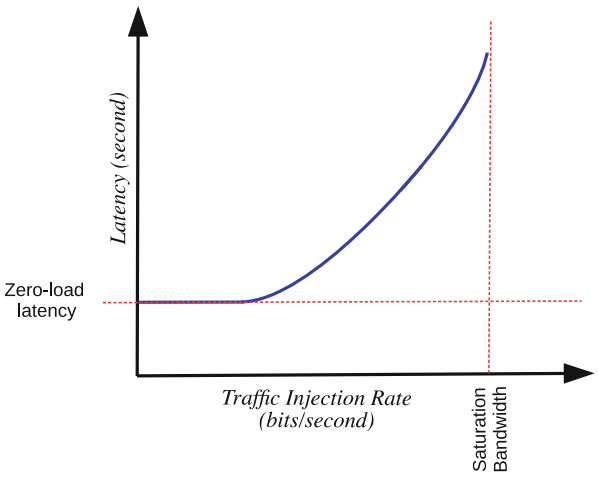


Fig. 15.8 Latency and throughput metrics for NoC

application-level metrics such as cache miss rate. This is the reason that recent works on NoC power optimization consider application-level metrics instead of network-level metrics [23, 25, 44].

15.3 Overview of Recent Academic and Commercial NoCs

Tilera iMesh on-chip interconnect network [103] is an example of NoC designed for homogeneous multi-core chips. The iMesh interconnect architecture has been used in the commercial TilePro64 multi-core chip. The latest 72-core Tilera GX chip [2] also uses the same NoC design. The proposed NoC architecture is different from other academic and commercially available on-chip architectures in terms of the number of physical NoC. iMesh provides five different physical NoC channels: two of these networks are used for memory access and management tasks, while the rest are user accessible. The motivation is that future integrated circuits will have enough silicon resources to integrate more than one NoC per chip.

The next-generation SPARC M7 processor combines three different NoC topologies for the on-chip interconnect [84], ring-based request network, point-to-point network for response messages, and mesh for data packets. The M7 processor uses a shared distributed cache model. The request ring network is used to broadcast the cache requests to all cores in the system. The point-to-point network is only used when adjacent cores share data. The mesh data network is built using ten port routers and is used primarily to stream data from memory controllers to local cache.

Anton2 is a special-purpose supercomputer for performing molecular dynamic simulations. The building block of the supercomputer is Anton ASIC which contains a number of special-purpose hardware units for numerical calculations [101]. Anton2 ASIC uses a 4×4 2D mesh for connecting on-chip components, whereas the chips are connected with each other using a 3D torus. The novel feature of the communication architecture is that the same set of routers are used for both intra- and inter-chip communication. About 10% of the total ASIC area is dedicated to on-chip communication.

SMART (single-cycle multi-hop asynchronous repeated traversal) NoC from MIT is another interesting low-latency on-chip communication architecture [56]. Authors observed that a data bit can travel 9–11 hops in a single clock cycle at 1 GHz for 45 nm silicon technology. Based on this observation, they propose a NoC architecture where data can be transferred across physically distant mesh nodes in a single cycle by bypassing multiple routers. This reduces the latency of multi-hop mesh NoC. The proposed architecture is reported to improve the performance of PARSEC and SPLASH-2 benchmarks by an average of 26% for private L2 cache and 49% for shared L2 cache architectures.

Æthereal [42] is probably one of the best-known academic NoC architecture. Even at the time of conception, Æthereal supported different IP communication protocols such as AXI and OCP. In addition to baseline best-effort NoC services, Æthereal also supports building predictable on-chip communication though time-division multiplexed circuits [43]. This enables building SoCs for system

with hard real-time performance requirements such as braking system in an automobile. A more comprehensive discussion on real-time NoCs is presented in ▶ Chap. 16, "NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications".

Intel used a mesh-based NoC for an 80-core TeraFlop experimental chip [46]. The mesh NoC uses five stage pipelined routers designed for 5 GHz frequency. This results in 1 ns per-hop latency. According to experiments conducted on the research chip, the NoC consumed about 28% of total chip power although it consumed 17% of total chip area.

Intel has also introduced a 48-core mesh NoC-based multi-core chip called single-chip cloud computer (SCC) [47]. The target frequency for the NoC was set at 2 GHz with 1.1 V supply voltage. The router is four-stage pipelined and uses virtual cut-through flow control. To mitigate the problem with higher NoC power from the previous 80-core chip, Intel opted for a DVFS scheme for NoC. The NoC was organized as a 6×6 mesh so that two compute cores share a single router. These techniques helped to reduce the share of NoC power to 10%.

There are two well-known commercial NoC IP providers, Sonics [4] and Arteris [7]. Both Sonics and Arteris provide various predesigned NoC IPs for commonly used processor IPs such as ARM and Xtensa. Furthermore, both of them provide design tools that can be used to optimize the NoC IP for various design objectives such as power, area, performance, quality of service (QoS), and reliability. It is anticipated that hardware developers will be using third-party NoCs to reduce both design cost and development time [20].

15.4 NoC Power Optimization

As more cores are integrated on a chip, the on-chip interconnect's complexity increases and so does its power. Computer architects always want to keep the on-chip interconnect's power low so that processing elements and memory hierarchy can use a larger share of power [17]. Although NoC provides the scalable communication for multi-core chips, it can consume valuable power. For example, Daya et al. [35] report that NoC in their 36-core SCORPIO experimental chip consumes 19% of the total chip power. Similarly NoC consumes 28% of total power in Intel's TeraFlop chip [46]. With limited power budgets constraining multi-core scaling [39], employing power-saving techniques for NoCs is an active research topic.

Over the years, various voltage-scaling-based solutions have been investigated for reducing NoC dynamic power. Shang et al. [92] presented the pioneering work on Dynamic Voltage and Frequency Scaling (DVFS) for on-chip links. The scheme uses usage history of links to predict their future utilization. Based on this prediction, a certain voltage and frequency (VF) level is selected for each link. Bogdan et al. [14] proposed a DVFS scheme for spatially and temporally varying workloads. The proposed DVFS scheme worked at the granularity of individual routers. Based on fractal modeling of workloads, the scheme selected an appropriate

VF level for each router. Mishra et al. [69] proposed per-router frequency tuning in response to changes in network workload to manage power and performance. Based on the optimization goal, the frequency of the routers is increased to relieve network congestions and improve performance, or the frequency is decreased to meet certain network power constraints. Ogras et al. [78] described a framework for fine-grain VF selection for VF-island-based NoCs. The scheme first statically allocates VF level to each NoC router and then uses the run-time network information for fine-tuning the assigned VF level. All these works base their DVFS schemes on network metrics such as packet latency, injection rate, queue occupancy, and network throughput. Researchers [26,44] argue that by neglecting higher-level performance metrics such as application execution time, these schemes can result in nonoptimal results. Therefore, work by Chen et al. [44] and Hesse and Enright [44] based their DVFS schemes on the actual execution delay faced by applications due to DVFS for NoCs. Zhan et al. [106] explored the problem of per-router DVFS schemes for streaming applications. They developed an analytical model to statically predict the effect of VF scaling on application throughput. Depending on application to core mapping, routers on the critical communication path are operated at lower frequency.

Since the ratio of leakage power to total chip power is increasing with transistor scaling, researchers have been exploring leakage power techniques. Through experiments with realistic benchmarks, Chen [23] reported that the router's static power share is 17.9% at 65 nm, 35.4% at 45 nm, and 47.7% at 32 nm. As discussed earlier in this chapter, power gating is often used to save the static power of on-chip components [48]. Soteriou and Peh [94] proposed power gating at the link level. They used the channel utilization history to select links that can be switched off with minimal impact on performance. As switching off some links results in irregular topologies, they proposed an adaptive routing algorithm to avoid the switched off links. Matsutani et al. [66] proposed adding low-frequency virtual channel buffers that can be power gated at run time to save leakage power. An ultra fine-grain power gating for NoC routers is proposed in [65]. They used special cell libraries where each Complementary Metal-Oxide-Semiconductor (CMOS) gate has a power gate signal. Therefore, the power for each router component such as input buffer, switch allocator, and crossbar can be individually turned on or off, based on the network activity. This helped to save leakage power by 78.9%. However, the main drawback of this technique is the complexity of including special power gate circuitry in each CMOS gate. The additional power circuitry also increases the router area. Kim et al. [55] proposed a fine-grain buffer management scheme for NoC routers. In the scheme, depending on network traffic load, the size of the input buffers was increased or decreased at run time. The unused buffer slots were power gated to save leakage power. However, this scheme only targeted buffer leakage power and neglected other router components. Parikh et al. [80] proposed performing power gating at the granularity of the *data-path segment* which consists of an input buffer, crossbar mux, and output link. In the router parking [88] scheme, routers for cores that are not under use are switched off, and the traffic is routed around the switched off routers through new routing paths that are calculated at run time.

The main problem with the previous proposals for power gating is the performance overhead due to wakeup latency. Chen and Pinkston [24] proposed overlaying a standard mesh NoC with a low-cost ring network. If a packet arrives at a router and the router is switched off, the packet is routed through the ring network which is always switched on. Das et al. [33] proposed to divide wider mesh NoCs into four smaller NoCs for efficiency. Additional NoCs are only switched on if the network load exceeds a certain limit and the unused NoC planes remain switched off. In this network connectivity is ensured.

As buffers in routers consume a considerable share of dynamic and leakage power, researchers have also explored using bufferless NoCs. The first effort in this direction was the BLESS router [71]. The idea is that instead of storing the flit in router, the flit is routed even in a direction that does not result in a minimal path. This means that even with some misrouting, the flit will eventually reach its destination. Therefore this scheme is applicable for routers with the number of output ports equal or higher than the number of input ports. The deadlock and livelock condition is avoided by allocating a channel to the oldest flit first. To improve some of the shortcomings of BLESS router, CHIPPER [40] was introduced. Both BLESS and CHIPPER show potential in reducing network power. However there are two concerns with bufferless NoCs. First, the deflection routing causes packets to take non-minimal routes resulting in longer packet delays. The effect of network delays is more significant for memory-intensive applications. Secondly, the bufferless routing results in out-of-order arrival of the flits which increases the complexity of the NI.

15.5 Communication-Aware Mapping

While designing an MPSoC-based system, it is not uncommon to have fixed specifications for the underlying hardware. Therefore, in such cases, it is the responsibility of the system programmer to use the available hardware resources efficiently. The situation is even more complex in the case of heterogeneous MPSoC architecture with NoC interconnect. Over the years, many techniques have been proposed for effectively mapping a given application on a target MPSoC architecture. Studying the mapping problem for NoC-based MPSoC architectures has been a key area of research. The unique data communication capabilities of NoC-based architectures demand a smart communication-aware mapping strategy [19, 28, 59, 73]. It is important to analyze both NoC's architectural properties and run-time network contention while estimating the run time of a given application. Embedded system designers are expected to tweak the application mapping to maximize the performance while meeting the hard time-to-market limits [99, 100]. There is no single technique to map applications on a given MPSoC. The mapping problem is further complicated due to the presence of many different MPSoC architectures. Therefore, the common practice is to rebuild the mapping strategy for every application-architecture combination [68, 74, 75, 97].

Application mapping was an area of interest for researchers working in parallel computing and supercomputing [15]. Their idea was to map the applications that

share the data as close as possible to reduce the network latency. This naive idea is also applicable in the case of CMPs where mapping the application has to take into account the underlying on-chip interconnect architecture. Similarly, the choice of shared memory architecture and message passing interface heavily influences the mapping algorithms.

The mapping solutions proposed in the literature can be broadly divided into two categories (1) static mapping and (2) dynamic run-time mapping. Each class of mapping algorithm has its advantages and disadvantages. Static mapping is useful when applications to be executed are known at design time [18, 77, 98, 108]. System designers can formulate an optimal or near-optimal solution for such scenarios. On the other hand, run-time mapping algorithms allocate resources to application on the fly. These algorithms give a better mapping solution in cases where the application behavior is unknown at design time or system characteristics such as network congestion can change rapidly [9, 10, 45].

In the case of embedded systems, applications to be executed are known at design time. Therefore, static mapping techniques result in better system performance [86]. Unfortunately, most of the mapping problems prove to be NP-hard problems with only near-optimal solution [93]. However, heuristic-based algorithms have also been proposed, which reduce the time required to solve these NP-hard problems [93, 97]. TDMA-based NoCs are used for predictable systems. The expected traffic for a given link can be estimated using analytical modeling or simulations. Through proposed algorithms, it is then possible to map a given processor-processor communication in an allotted time slot. This not only improves the network congestion but also makes the system predictable [102, 109].

Some researchers have provided solutions where the mapping and partitioning problem of NoC-based embedded systems are dealt with as a combined problem. Although these solutions require a very strict analytical model for application, the final software and hardware are highly optimized in area footprint, power, and performance [59, 60]. It has been observed that a generalized MPSoC architecture might eventually fail to cover the design requirements of a particular real-time embedded system. Therefore, the option of highly customized hardware for a particular set of applications has also been widely explored.

15.6 Application Specific Communication Architecture for MPSoCs

Over the years, researchers have analyzed that flat communication and memory architectures are not sufficient for designing high-performance application-specific MPSoC architectures. Moreover, the quest for low-power, low-cost embedded system has forced designers to optimize the system. Therefore, it is anticipated that highly optimized hardware and software designs will include customized memory and on-chip network designs in the future [99].

Customizing the communication architectures at system level and gate level not only improves the system performance; it also results in energy-efficient

design. By eliminating the logic overhead, the leakage and static power loss of the system can be significantly reduced [72]. It has been shown that some MPSoC applications actually do not need highly complex network architectures like NoC, and simple bus-based architecture can essentially meet the performance requirements. Therefore, it is important to avoid overkill by intelligently designing communication architectures that give optimal solutions without incurring area overhead [51].

The idea of application-specific communication architectures is more interesting in the case of NoC. Inherently, NoC architectures are highly customizable, and designers can tweak a variety of parameters of the architecture to meet performance, cost, and energy constraints. ×-pipes is perhaps the first project that provided a library-based approach to NoC design [96]. The selection of optimal on-chip communication architectures is nontrivial. Therefore, an effort has been made to introduce library-oriented design space exploration where the designer has several configurations to choose from. Designers can then use analytical modeling or simulations to select the best communication architectures [49]. The same idea has also been explored by Jeremy and Parameswaran [21] by developing a library of NoC components with the help of analytical power and performance modeling. Another interesting project that was targeted toward low-power NoC-based embedded MPSoC was AEthereal network on chip [42]. The significance of these projects is the flexibility provided to the designer to quickly explore the various possible optimizations for a given application. These optimizations can provide up to 12 times improvement in performance while reducing the area cost by 25 times when compared with flat communication architectures [51].

Bertozzi et al. [13] have proposed a NoC synthesis tools called *NetChip* based on ×-pipes [96] NoC library. Figure 15.9 shows an example on how application specifications are translated into NoC architecture. The first step is collect application and map it on multiple components (cores, memory, etc.). The next step is extract data bandwidth requirement from the application to core mapping. Based on communication requirement, tool automatically explores different NoC topologies and other associated architecture parameters such as routing scheme. In the final step, the tool automatically generates SystemC models for selected NoC components for simulations and synthesis.

Similarly, the option to integrate different communication architectures in a single design has also been studied extensively. Murali et al. [72] proposed a complete design method for application-specific crossbar synthesis. They use the traffic pattern of the application to design a communication architecture that is a combination of packet-oriented crossbars and bus. Bus-based architectures are simple but offer less performance. The crossbar-based architectures are more complex but provide better throughput. Therefore, Murali et al. combined the two communication architectures while keeping the gate cost low and fulfilling the performance requirements.

All the techniques referred to above improve the performance of on-chip communication in terms of aggregate on-chip bandwidth and average data latency, which are not suitable metrics for streaming applications executing on MPSoCs. In

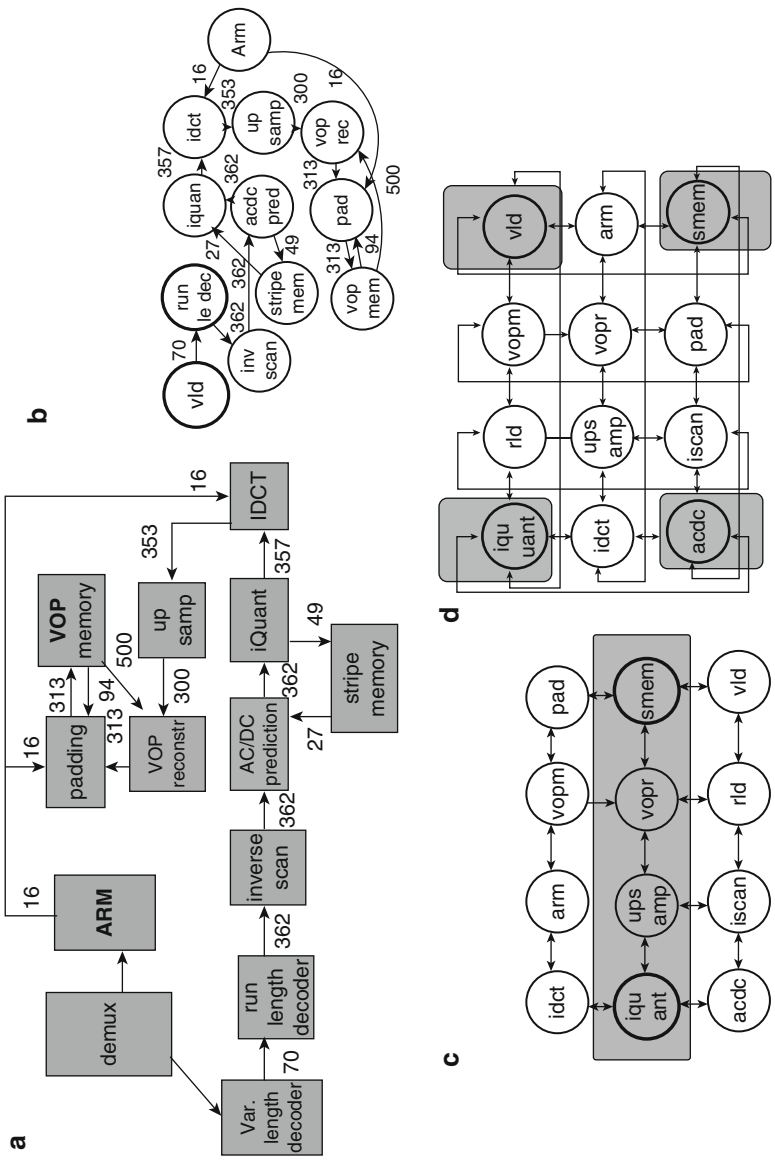


Fig. 15.9 Application-specific NoC design example from [13] for VOPD application. (**a**) VOPD block diagram, (**b**) VOPD graph with bandwidth requirements mentioned in mb/s, (**c**) mesh mapping, and (**d**) torus mapping

the case of streaming MPSoCs, application-level throughput or latency of the system is the most important performance metric [52]. Thus, it is important to incorporate system-level performance constraints in the design flow for an application-specific on-chip network [12].

15.7 Conclusion

In this chapter we introduced readers to some basic concepts of NoC architecture and motivated the use of NoC architecture for large-scale SoC designs. We presented examples of NoC designs from commercial chips and academia to show the current trends in NoC design. Given the increasing interest in reducing power consumption of SoC components, we presented various power optimization techniques proposed over the recent years. We then discussed some of the recent research work on optimizing performance using intelligent application mapping on MPSoCs. In the end, we explored how on-chip interconnect can be designed for application-specific MPSoCs.

References

1. ARM AMBA Interconnect Specification. http://www.arm.com/products/system-ip/amba-specifications.php
2. EZChip TileGX Multicore Architecture. http://www.tilera.com/products/?ezchip=585&spage=614
3. IBM CoreConnect Bus Technology. http://www.xilinx.com/products/intellectual-property/dr_pcentral_coreconnect.html
4. Sonics. http://www.sonicsinc.com/
5. Xtensa Processors. http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable
6. AMBA AXI and ACE Protocol Specification (2013) http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html
7. Arteris NoC (2015) http://www.arteris.com/
8. Aisopos K (2012) Fault tolerant architectures for on-chip networks. PhD Thesis, Princeton University
9. Al Faruque M, Krist R, Henkel J (2008) Adam: run-time agent-based distributed application mapping for on-chip communication. In: 45th ACM/IEEE design automation conference, DAC 2008, pp 760–765
10. Al Faruque MA, Krist R, Henkel J (2008) ADAM: run-time agent-based distributed application mapping for on-chip communication. In: Proceedings of the 45th IEEE/ACM design automation conference (DAC), pp 760–765. doi:10.1145/1391469.1391664
11. Becker DU (2012) Efficient microarchitecture for network-on-chip routers. Ph.D. thesis, Stanford University
12. Beraha R, Walter I, Cidon I, Kolodny A (2010) Leveraging application-level requirements in the design of a NoC for a 4g SoC – a case study. In: Design, automation test in Europe conference exhibition (DATE), pp 1408–1413. doi:10.1109/DATE.2010.5457033
13. Bertozzi D, Jalabert A, Murali S, Tamhankar R, Stergiou S, Benini L, De Micheli G (2005) NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. IEEE Trans Parallel Distrib Syst 16(2):113–129
14. Bogdan P, Marculescu R, Jain S, Gavila R (2012) An optimal control approach to power management for multi-voltage and frequency islands multiprocessor platforms under highly

variable workloads. In: 2012 sixth IEEE/ACM international symposium on networks on chip (NoCS), pp 35–42. doi:10.1109/NOCS.2012.32
15. Bokhari SH (1981) On the mapping problem. IEEE Trans Comput 30(3):207–214 doi:10.1109/TC.1981.1675756
16. Borkar S (2007) Thousand core chips: a technology perspective. In: Proceedings of the 44th design automation conference. ACM, pp 746–749
17. Borkar S, Chien AA (2011) The future of microprocessors. Commun ACM 54(5):67–77. doi:10.1145/1941487.1941507
18. Braun TD, Siegel HJ, Beck N, Bölöni LL, Maheswaran M, Reuther AI, Robertson JP, Theys MD, Yao B, Hensgen D et al (2001) A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J Parallel Distrib comput 61(6):810–837
19. Castrillon J, Tretter A, Leupers R, Ascheid G (2012) Communication-aware mapping of KPN applications onto heterogeneous MPSoCs. In: Proceedings of the 49th design automation conference, DAC'12. ACM, New York, pp 1266–1271. doi:10.1145/2228360.2228597
20. CHAN J (2007) Energy-aware synthesis for networks on chip architectures. Ph.D. thesis, UNSW
21. Chan J, Parameswaran S (2004) Nocgen: a template based reuse methodology for networks on chip architecture. In: 17th international conference on VLSI design, 2004. Proceedings. pp 717–720. doi:10.1109/ICVD.2004.1261011
22. Chao HL, Chen YR, Tung SY, Hsiung PA, Chen SJ (2012) Congestion-aware scheduling for NoC-based reconfigurable systems. In: Design, automation test in Europe conference exhibition (DATE), pp 1561–1566. doi:10.1109/DATE.2012.6176721
23. Chen L (2014) Design of low-power and resource-efficient on-chip networks. Ph.D. thesis, University of Southern California
24. Chen L, Pinkston TM (2012) Nord: node-router decoupling for effective power-gating of on-chip routers. In: Proceedings of the 45th annual IEEE/ACM international symposium on microarchitecture, MICRO'12. IEEE Computer Society, Washington, DC, pp 270–281. doi:10.1109/MICRO.2012.33
25. Chen X, Xu Z, Kim H, Gratz P, Hu J, Kishinevsky M, Ogras U (2012) In-network monitoring and control policy for DVFS of CMP networks-on-chip and last level caches. In: 2012 sixth IEEE/ACM international symposium on networks on chip (NoCS), pp 43–50. doi:10.1109/NOCS.2012.12
26. Chen X, Xu Z, Kim H, Gratz PV, Hu J, Kishinevsky M, Ogras U, Ayoub R (2013) Dynamic voltage and frequency scaling for shared resources in multicore processor designs. In: Proceedings of the 50th design automation conference, DAC'13. ACM, New York, pp 114:1–114:7. doi:10.1145/2463209.2488874
27. Cheng HY, Zhan J, Zhao J, Xie Y, Sampson J, Irwin MJ (2015) Core vs. uncore: the heart of darkness. In: 2015 52nd ACM/EDAC/IEEE design automation conference (DAC). IEEE, pp 1–6
28. Chou CL, Marculescu R (2008) Contention-aware application mapping for network-on-chip communication architectures. In: IEEE international conference on computer design, ICCD 2008, pp 164–169. doi:10.1109/ICCD.2008.4751856
29. Dally WJ, Towles B (2001) Route packets, not wires: on-chip interconnection networks. In: Design automation conference, 2001. Proceedings. IEEE, pp 684–689
30. Dally WJ, Towles BP (2004) Principles and practices of interconnection networks. Elsevier
31. Das R (2010) Application-aware on-chip networks. Ph.D. thesis, The Pennsylvania State University
32. Das R, Ausavarungnirun R, Mutlu O, Kumar A, Azimi M (2013) Application-to-core mapping policies to reduce memory system interference in multi-core systems. In: 2013 IEEE 19th international symposium on high performance computer architecture (HPCA2013), pp 107–118. doi:10.1109/HPCA.2013.6522311

33. Das R, Narayanasamy S, Satpathy SK, Dreslinski RG (2013) Catnap: energy proportional multiple network-on-chip. In: Proceedings of the 40th annual international symposium on computer architecture, ISCA'13. ACM, New York, pp 320–331. doi:10.1145/2485922.2485950
34. Daveau JM, Ismail TB, Jerraya AA (1995) Synthesis of system-level communication by an allocation-based approach. In: Proceedings of the 8th international symposium on system synthesis. ACM, pp 150–155
35. Daya BK, Chen CHO, Subramanian S, Kwon WC, Park S, Krishna T, Holt J, Chandrakasan AP, Peh LS (2014) Scorpio: a 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering. In: 2014 ACM/IEEE 41st international symposium on computer architecture (ISCA). IEEE, pp 25–36
36. Jerraya AA, Wolf W (2005) Multiprocessor systems-on-chips. The Morgan Kaufmann series in systems on silicon. Morgan Kaufmann. ISBN:0-12385-251-X
37. Dennard RH, Gaensslen FH, Rideout VL, Bassous E, LeBlanc AR (1974) Design of Ion-implanted MOSFET's with very small physical dimensions. IEEE J Solid-State Circuits 9(5):256–268
38. Dimitrakopoulos G, Psarras A, Seitanidis I (2015) Microarchitecture of network-on-chip routers. Springer
39. Esmaeilzadeh H, Blem E, St.Amant R, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. In: 38th annual international symposium on computer architecture (ISCA), pp 365–376
40. Fallin C, Craik C, Mutlu O (2011) Chipper: a low-complexity bufferless deflection router. In: 2011 IEEE 17th international symposium on high performance computer architecture (HPCA), pp 144–155. doi:10.1109/HPCA.2011.5749724
41. Gasteier M, Glesner M (1996) Bus-based communication synthesis on system-level. In: 9th international symposium on system synthesis, 1996. Proceedings. IEEE, pp. 65–70
42. Goossens K, Hansson A (2010) The AEthereal network on chip after ten years: goals, evolution, lessons, and future. In: Proceedings of the 47th design automation conference, DAC'10. ACM, New York, pp 306–311. doi:10.1145/1837274.1837353
43. Hansson A, Wiggers M, Moonen A, Goossens K, Bekooij M (2008) Applying dataflow analysis to dimension buffers for guaranteed performance in networks on chip. In: Proceedings of international symposium on networks on chip (NOCS). IEEE Computer Society, Washington, DC, pp 211–212
44. Hesse R, Jerger NE (2015) Improving DVFS in NoCs with coherence prediction. In: 2015 9th IEEE/ACM international symposium on networks on chip (NoCS)
45. Hölzenspies PKF, Hurink JL, Kuper J, Smit GJM (2008) Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSoC). In: Proceedings of the conference on design, automation and test in Europe, DATE'08. ACM, New York, pp 212–217. doi:10.1145/1403375.1403427
46. Hoskote Y, Vangal S, Singh A, Borkar N, Borkar S (2007) A 5-GHz mesh interconnect for a TeraFlops processor. IEEE Micro 27(5):51–61
47. Howard J, Dighe S, Vangal S, Ruhl G, Borkar N, Jain S, Erraguntla V, Konow M, Riepen M, Gries M, Droege G, Lund-Larsen T, Steibl S, Borkar S, De V, Van Der Wijngaart R (2011) A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling. IEEE J Solid-State Circuits 46(1):173–183. doi:10.1109/JSSC.2010.2079450
48. Hu Z, Buyuktosunoglu A, Srinivasan V, Zyuban V, Jacobson H, Bose P (2004) Microarchitectural techniques for power gating of execution units. In: Proceedings of the 2004 international symposium on low power electronics and design. ACM, pp 32–37
49. Huang PK, Hashemi M, Ghiasi S (2008) System-level performance estimation for application-specific MPSoC interconnect synthesis. In: Symposium on application specific processors, SASP 2008, pp 95–100. doi:10.1109/SASP.2008.4570792

50. Iordanou C, Soteriou V, Aisopos K (2014) Hermes: architecting a top-performing fault-tolerant routing algorithm for networks-on-chips. In: 2014 IEEE 32nd international conference on computer design (ICCD). IEEE, pp 424–431
51. Jan Y, Jóźwiak L (2012) Communication and memory architecture design of application-specific high-end multiprocessors. VLSI Des 2012:12:12–12:12. doi:10.1155/2012/794753
52. Javaid H, He X, Ignjatovic A, Parameswaran S (2010) Optimal synthesis of latency and throughput constrained pipelined MPSoCs targeting streaming applications. In: 2010 IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 75–84
53. Jerraya A, Wolf W (2005) Multiprocessor systems-on-chips. Electronics & electrical. Morgan Kaufmann. http://books.google.com.au/books?id=7i9Z69lrYBoC
54. Kermani P, Kleinrock L (1979) Virtual cut-through: a new computer communication switching technique. Comput Netw (1976) 3(4):267–286
55. Kim G, Kim J, Yoo S (2011) Flexibuffer: reducing leakage power in on-chip network routers. In: 2011 48th ACM/EDAC/IEEE design automation conference (DAC), pp 936–941
56. Krishna T, Chen CHO, Kwon WC, Peh LS (2014) Smart: single-cycle multihop traversals over a shared network on chip. IEEE Micro 34(3):43–56
57. Kumar R, Zyuban V, Tullsen DM (2005) Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling. In: 32nd international symposium on computer architecture, ISCA'05. Proceedings. IEEE, pp 408–419
58. Kumar S, Jantsch A, Soininen JP, Forsell M, Millberg M, Oberg J, Tiensyrja K, Hemani A (2002) A network on chip architecture and design methodology. In: IEEE computer society annual symposium on VLSI, 2002. Proceedings. IEEE, pp 105–112
59. Le Beux S, Bois G, Nicolescu G, Bouchebaba Y, Langevin M, Paulin P (2010) Combining mapping and partitioning exploration for NoC-based embedded systems. J Syst Archit 56(7):223–232. doi:10.1016/j.sysarc.2010.03.005
60. Le Beux S, Nicolescu G, Bois G, Bouchebaba Y, Langevin M, Paulin P (2009) Optimizing configuration and application mapping for MPSoC architectures. In: NASA/ESA conference on adaptive hardware and systems, AHS 2009, pp 474–481. doi:10.1109/AHS.2009.35
61. Lee HG, Chang N, Ogras UY, Marculescu R (2007) On-chip communication architecture exploration: a quantitative evaluation of point-to-point, bus, and network-on-chip approaches. ACM Trans Des Autom Electron Syst (TODAES) 12(3):23
62. Ly D, Saldana M, Chow P (2009) The challenges of using an embedded MPI for hardware-based processing nodes. In: International conference on field-programmable technology, FPT 2009, pp 120–127. doi:10.1109/FPT.2009.5377688
63. Ma S, Enright Jerger N, Wang Z (2011) DBAR: an efficient routing algorithm to support multiple concurrent applications in networks-on-chip. In: Proceedings of the 38th annual international symposium on computer architecture, ISCA'11. ACM, New York, pp 413–424. doi:10.1145/2000064.2000113
64. Mahadevan S, Angiolini F, Storgaard M, ndahl Olsen RG, SparsøJ (2005) A network traffic generator model for fast network-on-chip simulation. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE). Mahadevan05.pdf
65. Matsutani H, Koibuchi M, Ikebuchi D, Usami K, Nakamura H, Amano H (2011) Performance, area, and power evaluations of ultrafine-grained run-time power-gating routers for CMPs. IEEE Trans Comput-Aided Des Integr Circuits Syst 30(4):520–533. doi:10.1109/T-CAD.2011.2110470
66. Matsutani H, Koibuchi M, Wang D, Amano H (2008) Adding slow-silent virtual channels for low-power on-chip networks. In: Second ACM/IEEE international symposium on networks-on-chip, NoCS 2008, pp 23–32. doi:10.1109/NOCS.2008.4492722
67. Medardoni S (2009) Driving the network-on-chip revolution to remove the interconnect bottleneck in nanoscale multi-processor systems-on-chip. Ph.D. thesis, Università degli studi di Ferrara
68. Mirzoyan D, Akesson B, Goossens K (2014) Process-variation aware mapping of best-effort and real-time streaming applications to MPSoCs. ACM Trans Embed Comput Syst (TECS) 13(61):61:1–61:24

69. Mishra A, Das R, Eachempati S, Iyer R, Vijaykrishnan N, Das C (2009) A case for dynamic frequency tuning in on-chip networks. In: 42nd annual IEEE/ACM international symposium on microarchitecture, MICRO-42, pp 292–303
70. Mishra AK, Mutlu O, Das CR (2013) A heterogeneous multiple network-on-chip design: an application-aware approach. In: Proceedings of the 50th annual design automation conference, DAC'13. ACM, New York, pp 36:1–36:10. doi:10.1145/2463209.2488779
71. Moscibroda T, Mutlu O (2009) A case for bufferless routing in on-chip networks. SIGARCH Comput Archit News 37(3):196–207. doi:10.1145/1555815.1555781
72. Murali S, Benini L, De Micheli G (2007) An application-specific design methodology for on-chip crossbar generation. IEEE Trans Comput-Aided Des Integr Circuits Syst 26(7):1283–1296. doi:10.1109/TCAD.2006.888284
73. Murali S, Coenen M, Radulescu A, Goossens K, De Micheli G (2006) Mapping and configuration methods for multi-use-case networks on chips. In: Asia and South Pacific conference on design automation, 6pp. doi:10.1109/ASPDAC.2006.1594673
74. Murali S, Coenen M, Rădulescu A, Goossens K, De Micheli G (2006) A methodology for mapping multiple use-cases on to networks on chip. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE). European Design and Automation Association, 3001 Leuven, pp 118–123
75. Nejad AB, Goossens K, Walters J, Kienhuis B (2009) Mapping KPN models of streaming applications on a network-on-chip platform. In: Proceedings of annual workshop on circuits, systems and signal processing (ProRisc)
76. Nesson T, Johnsson SL (1995) ROMM routing on mesh and torus networks. In: Proceedings of the seventh annual ACM symposium on parallel algorithms and architectures. ACM, pp 275–287
77. Nikitin N, Cortadella J (2012) Static task mapping for tiled chip multiprocessors with multiple voltage islands. In: Proceedings of the 25th international conference on architecture of computing systems (ARCS), pp 50–62
78. Ogras UY, Marculescu R, Marculescu D, Jung EG (2009) Design and management of voltage-frequency island partitioned networks-on-chip. IEEE Trans Very Large Scale Integr (VLSI) Syst 17(3):330–341
79. Palesi M, Holsmark R, Kumar S, Catania V (2006) A methodology for design of application specific deadlock-free routing algorithms for NoC systems. In: Proceedings of the 4th international conference on hardware/software codesign and system synthesis. ACM, pp 142–147
80. Parikh R, Das R, Bertacco V (2014) Power-aware NoCs through routing and topology reconfiguration. In: 2014 51st ACM/EDAC/IEEE design automation conference (DAC). IEEE, pp 1–6
81. Park S, Krishna T, Chen CH, Daya B, Chandrakasan A, Peh LS (2012) Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45 nm soi. In: Proceedings of the 49th annual design automation conference, DAC'12. ACM, New York, pp 398–405. doi:10.1145/2228360.2228431
82. Passas G, Katevenis M, Pnevmatikatos D (2012) Crossbar NoCs are scalable beyond 100 nodes. IEEE Trans Comput-Aided Des Integr Circuits Syst 31(4):573–585
83. Peh L, Jerger N (2009) On-chip networks (synthesis lectures on computer architecture). Morgan and Claypool, San Rafael
84. Phillips S (2014) M7: next generation sparc. In: Hot chips: a symposium on high performance chips
85. Rădulescu A, Dielissen J, Goossens K, Rijpkema E, Wielage P (2004) An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), vol 2. IEEE Computer Society, Washington, DC, pp 878–883
86. Salamy H, Ramanujam J (2012) An effective solution to task scheduling and memory partitioning for multiprocessor system-on-chip. IEEE Trans Comput-Aided Des Integr Circuits Syst 31(5):717–725. doi:10.1109/TCAD.2011.2181848

87. Salminen E, Lahtinen V, Kuusilinna K, Hamalainen T (2002) Overview of bus-based system-on-chip interconnections. In: IEEE international symposium on circuits and systems, ISCAS 2002, vol 2. IEEE, pp II–372
88. Samih A, Wang R, Krishna A, Maciocco C, Tai C, Solihin Y (2013) Energy-efficient interconnect via router parking. In: 2013 IEEE 19th international symposium on high performance computer architecture (HPCA2013), pp 508–519. doi:10.1109/HPCA.2013.6522345
89. Seiculescu C, Murali S, Benini L, De Micheli G (2009) NoC topology synthesis for supporting shutdown of voltage islands in SoCs. In: 46th ACM/IEEE design automation conference, DAC'09, pp 822–825
90. Seiculescu C, Murali S, Benini L, De Micheli G (2010) A method to remove deadlocks in networks-on-chips with wormhole flow control. In: Proceedings of the conference on design, automation and test in Europe. European Design and Automation Association, pp 1625–1628
91. Shafik RA, Rosinger P, Al-Hashimi BM (2008) MPEG-based performance comparison between network-on-chip and AMBA MPSoC. In: 11th IEEE workshop on design and diagnostics of electronic circuits and systems (DDECS) 2008, pp 1–6
92. Shang L, Peh LS, Jha N (2003) Dynamic voltage scaling with links for power optimization of interconnection networks. In: The ninth international symposium on high-performance computer architecture, HPCA-9 2003. Proceedings, pp 91–102. doi:10.1109/HPCA.2003.1183527
93. Singh AK, Shafique M, Kumar A, Henkel J (2013) Mapping on multi/many-core systems: survey of current and emerging trends. In: Proceedings of the 50th IEEE/ACM design automation conference (DAC), pp 1:1–1:10. doi:10.1145/2463209.2488734
94. Soteriou V, Peh LS (2007) Exploring the design space of self-regulating power-aware on/off interconnection networks. IEEE Trans Parallel Distrib Syst 18(3):393–408. doi:10.1109/TPDS.2007.43
95. Srinivasan K, Chatha KS (2006) A low complexity heuristic for design of custom network-on-chip architectures. In: Proceedings of the conference on design, automation and test in Europe: Proceedings. European Design and Automation Association, pp 130–135
96. Stergiou S, Angiolini F, Carta S, Raffo L, Bertozzi D, De Micheli G (2005) Xpipes lite: a synthesis oriented design library for networks on chips. In: Design, automation and test in Europe, 2005. Proceedings, vol 2, pp 1188–1193. doi:10.1109/DATE.2005.1
97. Stuijk S (2007) Predictable mapping of streaming applications on multiprocessors. Ph.D. thesis, Eindhoven University of Technology
98. Stuijk S, Basten T, Geilen M, Corporaal H (2007) Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In: 44th ACM/IEEE design automation conference, DAC'07, pp 777–782
99. Teich J (2012) Hardware/software codesign: the past, the present, and predicting the future. Proc IEEE 100(Special Centennial Issue), 1411–1430. doi:10.1109/JPROC.2011.2182009
100. Thiele L, Bacivarov I, Haid W, Huang K (2007) Mapping applications to tiled multiprocessor embedded systems. In: International conference on application of concurrency to system design, pp 29–40. doi:10.1109/ACSD.2007.53
101. Towles B, Grossman J, Greskamp B, Shaw DE (2014) Unifying on-chip and inter-node switching within the Anton2 network. In: 2014 ACM/IEEE 41st international symposium on computer architecture (ISCA). IEEE, pp 1–12
102. Weichslgartner A, Gangadharan D, Wildermann S, GlaßM, Teich J (2014) DAARM: design-time application analysis and run-time mapping for predictable execution in many-core systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 34:1–34:10. doi:10.1145/2656075.2656083
103. Wentzlaff D, Griffin P, Hoffmann H, Bao L, Edwards B, Ramey C, Mattina M, Miao CC, Brown III JF, Agarwal A (2007) On-chip interconnection architecture of the tile processor. IEEE Micro 27(5):15–31. doi:10.1109/MM.2007.89

104. Yoo J, Lee D, Yoo S, Choi K (2007) Communication architecture synthesis of cascaded bus matrix. In: Asia and South Pacific, design automation conference, ASP-DAC'07. IEEE, pp 171–177
105. Yoo S, Rha K, Cho Y, Kung J, Choi K (2002) Performance estimation of multiple-cache IP-based systems: case study of an interdependency problem and application of an extended shared memory model. In: International workshop on hardware/software codesign (CODES)
106. Zhan J, Stoimenov N, Ouyang J, Thiele L, Narayanan V, Xie Y (2013) Designing energy-efficient NoC for real-time embedded systems through slack optimization. In: Proceedings of the 50th annual design automation conference. ACM, p 37
107. Zhang YP, Jeong T, Chen F, Wu H, Nitzsche R, Gao GR (2006) A study of the on-chip interconnection network for the IBM Cyclops64 multi-core architecture. In: IEEE parallel and distributed processing symposium (IPDPS)
108. Zhu J, Sander I, Jantsch A (2010) Constrained global scheduling of streaming applications on MPSoCs. In: 2010 15th Asia and South Pacific design automation conference (ASP-DAC), pp 223–228. doi:10.1109/ASPDAC.2010.5419892
109. Zimmer C, Mueller F (2012) Low contention mapping of real-time tasks onto TilePro64 core processors. In: Proceedings of the 2012 IEEE 18th real time and embedded technology and applications symposium, RTAS'12. IEEE Computer Society, Washington, DC, pp 131–140. doi:10.1109/RTAS.2012.36

NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications

16

Kees Goossens, Martijn Koedam, Andrew Nelson, Shubhendu Sinha, Sven Goossens, Yonghui Li, Gabriela Breaban, Reinier van Kampenhout, Rasool Tavakoli, Juan Valencia, Hadi Ahmadi Balef, Benny Akesson, Sander Stuijk, Marc Geilen, Dip Goswami, and Majid Nabi

Abstract

In this chapter we define what a mixed-time-criticality system is and what its requirements are. After defining the concepts that such systems should follow, we described CompSOC, which is one example of a mixed-time-criticality platform. We describe, in detail, how multiple resources, such as processors, memories, and interconnect, are combined into a larger hardware platform, and especially how they are shared between applications using different arbitration schemes. Following this, the software architecture that transforms the single hardware platform into multiple virtual execution platforms, one per application, is described.

Acronyms

AHB	Advanced High-performance Bus
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BD	Budget Descriptor
CCSP	Credit-Controlled Static Priority
CDC	Clock Domain Crossing
CM	Communication Memory
DLMB	Data Local Memory Bus
DMA	Direct Memory Access

K. Goossens (✉) • M. Koedam • A. Nelson • S. Sinha • S. Goossens • Y. Li • G. Breaban • R. van Kampenhout • R. Tavakoli • J. Valencia • H.A. Balef • B. Akesson • S. Stuijk • M. Geilen • D. Goswami • M. Nabi
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: k.g.w.goossens@tue.nl; m.l.p.j.koedam@tue.nl; a.t.nelson@tue.nl; s.sinha@tue.nl; s.l.m.goossens@tue.nl; yonghui.li@tue.nl; g.breaban@tue.nl; j.r.v.kampenhout@tue.nl; r.tavakoli@tue.nl; j.valencia@tue.nl; h.ahmadi.balef@tue.nl; k.b.akesson@tue.nl; s.stuijk@tue.nl; m.c.w.geilen@tue.nl; d.goswami@tue.nl; m.nabi@tue.nl

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_17

DMAMEM	DMA Memory
DMEM	Data Memory
DRAM	Dynamic Random-Access Memory
ELF	Executable and Linkable Format
ET	Execution Time
ETSCH	Extended TSCH
FBSP	Frame-Based Static Priority
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GALS	Globally Asynchronous Locally Synchronous
ILMB	Instruction Local Memory Bus
IMEM	Instruction Memory
I/O	Input/Output
IP	Intellectual Property
IPB	Intellectual Property Block
KPN	Kahn Process Network
MAC	Media Access Control
MMIO	Memory-Mapped I/O
MPSoC	Multi-Processor System-on-Chip
NI	Network Interface
NoC	Network-on-Chip
PLB	Processor Local Bus
RR	Round Robin
RT	Response Time
RTOS	Real-Time Operating System
SI	Scheduling Interval
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
TDM	Time-Division Multiplexing
TFT	Thin-Film Transistor
TIFU	Timer, Interrupt, and Frequency Unit
TSCH	Time-Synchronised Channel Hopping
TTA	Transport-Triggered Architecture
UART	Universal Asynchronous Receiver/Transmitter
VEP	Virtual Execution Platform
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

Contents

16.1 Introduction and Requirements

Electronics is pervasive: it enables applications and functions that we have come to expect from appliances as diverse as cars, planes, mobile phones, fridges, and light switches. At the heart of these appliances are Systems-on-Chips (SoCs) that execute the applications. Traditionally, each SoC executed one application, but to reduce cost, multiple applications are increasingly executed on the same SoC. Different applications have different requirements, such as high performance, varying degrees of real time, and safety. In this chapter, we focus on *mixed-time-criticality systems*, i.e., those featuring a combination of applications with and without real-time requirements, respectively. We do not consider other, equally important, criticality aspects, such as safety or resilience. Applications with real-time requirements can be as diverse as motor management, braking, or vehicle stability in a car or wired and wireless communication stacks in mobile phones or computers. This type of applications should always finish computations before given deadlines to ensure correctness and/or safety. In contrast, applications such as a graphical user interface or file management should be responsive to the user but do not have real-time requirements. Audio and video analysis, e.g., for night vision in a car, and media playback are an intermediate category where deadlines should generally be met but may occasionally be missed.

In this chapter, we will define a *mixed-criticality platform*, i.e., a general template, for systems that execute multiple applications with different time criticalities. Given the examples of (non)-real-time applications, we can state the requirements for such platforms.

1. *Guaranteed worst-case performance for real-time applications*. We call this *predictability*, by which we mean that the Worst-Case Response Time (WCRT) of an application can be computed at design time. The Worst-Case Response Time is what has to be guaranteed, but it is usually advantageous to additionally minimize the actual Response Time (RT).
2. *As good as possible, actual-case performance for non-real-time applications.* Unlike real-time applications, the worst-case response time is not relevant and may not even exist. The average or actual response time should therefore be minimized instead.
3. *The absence of interference between applications*. The guaranteed or best-possible performance has to be guaranteed for each real-time and non-real-time application, even though they share the same platform (resources). To be able to do this independently per application, we additionally require that the *actual* execution time of an application is independent of other applications. It then follows that worst-case execution and response times are independent too. We call this *composability*. It helps to isolate (software) faults of applications, increasing robustness. Perhaps more importantly, it allows each application to be developed, tested, and deployed independently, which is required for certification. It also eases upgrading part of a system, without having to retest or recertify the system as a whole.

In this chapter, we describe the CompSOC platform, which is an example of a mixed-criticality platform that meets all the requirements. First, we define the concepts that underpin the CompSOC platform but which are common to many of mixed-criticality platforms (discussed in Sect. 16.6). We describe the hardware architecture of CompSOC in Sect. 16.3 and the software architecture in Sect. 16.4. An example usage of the CompSOC platform is given in Sect. 16.5. We conclude in Sect. 16.7.

16.2 Concepts for a Mixed-Time-Criticality Platform

First we need to introduce some terminology. An *application* is an independent (possibly cyclic) graph of communicating tasks. Tasks use platform *resources*, such as processors, memories, Direct Memory Accesss (DMAs), and interconnect. We model this with the notion of a *requestor*, e.g., a software task requesting computation from a processor, communication from the Network-on-Chip (NoC), or storage from a memory. Each requestor uses only one resource. A requestor generates *requests*, such as computing a function or a memory transaction. As illustrated in Fig. 16.1a, resources serve requestors in discrete uninterrupted units called *service units*, and a request consists of one or more service units (possibly infinitely many). The execution of a service unit by a resource takes an actual Execution Time (ET), which is less than or equal to its Worst-Case Execution Time (WCET) if it exists (i.e., it is finite). The WCET of a requestor is equal to the largest WCET of its requests. A resource is *predictable* if all service units have

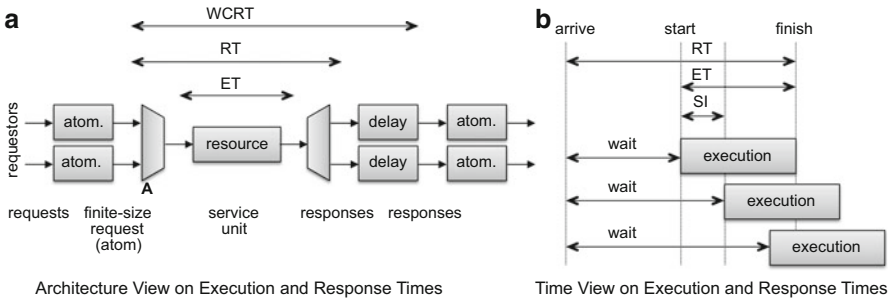


Fig. 16.1 Terminology. (**a**) Request, response, service unit, execution and response time, arbitration (*A* in the figure), and resource. The atom and atomizer will be discussed in Sect. 16.3.2. (**b**) Arrival, start, finish, and scheduling interval

a WCET. A resource is *composable* if the response and the execution time of every request of an application do not change when also executing any number of requests of different applications on the resource.

A resource is *shared* if it executes requests of multiple requestors, which may belong to different applications. The execution time of a service unit does not take into account waiting time for other service units from the same or different requestors. Instead, this is captured by its actual response time and its Worst-Case Response Time (WCRT). (As before, we say the WCRT exists if it is finite. The WCRT of a requestor is equal to the largest WCRT of its requests.) The waiting time depends on the arbitration employed to determine the order of execution of the service units of the different requestors, shown by A in Fig. 16.1a. Possible arbitration policies include Round Robin (RR), Time-Division Multiplexing (TDM), Credit-Controlled Static Priority (CCSP) [5], Frame-Based Static Priority (FBSP) [2], etc., each with their own characteristics and (dis)advantages which we will discuss later. An arbiter is *predictable* if all requests have a WCRT (i.e., it is finite), assuming that their ETs are finite. (Thus, even with a predictable arbiter, a request may have no (i.e., infinite) WCRT if the resource is unpredictable.) Similarly, an arbiter is *composable* if the response and the response time of every request of an application do not change when also executing any number of requests of different applications on the resource.

The resource requirements of a requestor are specified using a *budget*. A requestor can use a resource only after its budget has been reserved. A resource may be *idle*, i.e., no service unit is executed. This may occur if not all of its capacity has been reserved or if a requestor does not use all of its reservation. Arbitration is *work conserving* if the resource is not idle whenever a service unit is waiting to be executed.

Figure 16.1b illustrates the Scheduling Interval (SI) of a resource as the time between accepting successive service units, and its reciprocal (service units per second) defines the throughput. Resources are often pipelined, and then the reciprocal of the scheduling interval defines a higher throughput than the reciprocal

of the execution time. From the response times of individual requests (executing on processors, interconnect, or accessing memories), it must be possible to compute the performance (worst-case response time, throughput, etc.) of a real-time application as a whole. As an example, [37] illustrates how to do this using the dataflow model of computation.

Given the terminology, these are the seven concepts on which we base our mixed-time-criticality platform:

1. **Budgets.** Reserving part of a resource for a requestor belonging to an application according to its budget results in a *virtual resource*. Only then can a requestor use the resource.
2. **Predictability.** *Arbitration between requestors of a real-time application must be predictable*. Predictable arbitration ensures that a virtual resource offered by a single resource has a minimum guaranteed performance as specified in the budget. Arbitration is preferably work conserving such that requestors of an application can use each other's unused capacity.
3. **Composability.** *Arbitration between requestors of different applications must be composable*: the behavior of one application must not be affected by the behavior of other applications. Composable arbitration ensures that (the performance of) a virtual resource is independent of other virtual resources on the same resource. This implies that the arbitration cannot be work conserving because then (variable) execution times of one application could affect the response time of other applications.
4. **Scalability.** *Decouple resources* as much as possible. Logically, this means that *each resource arbitrates locally*, with a suitable service unit and arbiter to enforce the reserved budgets. This disallows synchronization hardware, such as mutexes, locks, and semaphores, as we will discuss in Sect. 16.3.8. Physically, decoupling requires that all hardware Intellectual Property Blocks (IPBs) use independent clocks because in modern Multi-Processor Systems-on-Chips (MPSoCs), it is no longer possible to use a single clock. This is called the *Globally Asynchronous Locally Synchronous (GALS) design style*. Logical and physical decoupling improve scalability by avoiding centralized and tightly synchronized architectures.
5. **Finite scheduling interval.** As a consequence of Bullet 3, resources that are shared by multiple applications must ensure that no (service unit of a) requestor can indefinitely block others from using the resource. This is achieved with a finite scheduling interval, implemented either by chopping (possibly infinitely large) requests into service units with a finite WCET or else by preempting the resource within a finite time.
6. **Efficient arbitration.** If possible, *avoid arbitration* on a resource; e.g., a DMA is cheap enough to just replicate. Otherwise, at least *one level of composable arbitration* is required to arbitrate between requests belonging to different applications as well as between requests of the same application. (If a resource is used by only one application, then only one level of arbitration is needed, which must be predictable at most.) Preferably, *two levels of arbitration* are used: the first level to separate different applications and the second level to separate

requests of the same application. The former must be composable, and the latter predictable. For each resource, we will discuss in depth what kind of arbitration it can efficiently support.

7. **Efficient resource sharing.** As an optimization, *the scheduling interval should be as small as possible.* This allows interleaving the service of requestors as finely as possible and reduces response times [57]. How small the scheduling interval (and service units) can be depends strongly on the resource.

The above ingredients can be combined in the concept of a *Virtual Execution Platform (VEP) per application*. This means that an application's budget is reserved on all of the platform resources it requires, creating a smaller virtual platform on which it executes. A VEP is composable, i.e., independent of other VEPs and applications running therein. Within a VEP, an application may use its own programming model, arbitration, and so on, as long as it complies with the requirements outlined above. The CompSOC platform is an operational prototype implementing the concepts just described. It is our running example, as we go through the details in the remainder of this chapter. In the next section, we describe the hardware components of the CompSOC platform, followed by the software stack in Sect. 16.4. We discuss related work in Sect. 16.6 before concluding in Sect. 16.7.

16.3 Hardware Architecture

MPSoCs contain multiple processors with local and shared memories. The processor's local memories are always on-chip Static Random-Access Memory (SRAM), close to the processor. Nonlocal memories shared between processors may be on-chip SRAM but often include off-chip Dynamic Random-Access Memory (DRAM). The latter has a much larger capacity (number of bits) than the on-chip memory, but at the cost of a longer execution time. Processors reach shared memories using a communication infrastructure, which is increasingly a NoC. A NoC is a miniature version of the Internet in the sense that communication is concurrent, is distributed, and is either packet based or circuit switched. In this section, we introduce the CompSOC hardware platform. It not only fits the generic MPSoC description, it also addresses all the requirements listed in Sect. 16.1. As a result, it can run multiple applications of different criticalities at the same time.

The CompSOC platform consists of multiple tiles interconnect by a NoC. Tile types are master tiles, slave tiles, or a mix and include processor tiles, memory tiles, peripheral tiles, etc. In the following sections, we discuss each component in turn. Regarding terminology, an IPB, such as a memory, processor, or DMA, may have zero or more master and slave ports. Master ports initiate requests, i.e., read and write transactions, using a standard communication protocol, such as Processor Local Bus (PLB), Advanced High-performance Bus (AHB), or Advanced eXtensible Interface (AXI). Slave ports accept requests, and the slave executes them, possibly returning a response. In the following, we will shorten "master (slave) port on an IPB" to just "master (slave) IPB."

In the following sections, we will introduce master and slave IPBs, followed by processor tile that is a hybrid master-slave IPB. Described next is how all IPBs use distributed shared memory to communicate and how this is implemented by the NoC and the memory map. Finally, we will discuss atomic and synchronized communication.

16.3.1 Generic Master IP Block

A master IPB initiates read or write transactions that are to be transported by the NoC to a slave IPB for execution. The generic architecture of a master IPB is shown in the middle of Fig. 16.2. As will be explained in more detail in Sect. 16.3.6, ports on the NoC are connected pair-wise: only one slave IPB can be reached from a master single port. For this reason, a master IPB requires a NoC port for each slave it communicates with. The multiplexer labeled M3 determines to which slave a transaction is bound, depending on the transaction address. Any responses are interleaved in the order of the requests by the (de)multiplexer M3, before being returned to the master IPB [21]. The multiplexer and the master IPB may be programmed using the rightmost slave port on the tile.

16.3.2 Generic Slave IP Block and Memory Tile

Memory tiles only contain one slave resource that accepts requests from multiple requestors. Although we describe the architecture of the memory tile, it is essentially the same as that of any generic shared slave IPB. Each requestor has a dedicated NoC port on the tile, as illustrated by the general slave tile on the left in Fig. 16.2. Requests take the form of transactions, using, e.g., the PLB or AXI protocol. It is possible to read or write one or more words of 4 bytes, with a byte mask applied to each word in the transaction. Note, however, that in theory transactions can be infinitely long and that they can take a long (or even infinite) time to arrive. It may thus not be possible to buffer an entire transaction (request). For this reason, an atomizer chops incoming requests into complete fixed-size aligned requests (also called *atoms*). Note that even though these requests have a finite size, their execution time may be infinite; consider, for example, a `while(1);` request.

16.3.2.1 Arbitration

When a service unit is complete, it is ready to be scheduled by the arbiter, according to some policy. If the slave is only used by a single non-real-time application, then any arbitration policy may be used; if it is a real-time application, then the policy must be predictable for a WCRT, such as RR, TDM, or CCSP. However, it is likely that slaves such as SRAM are used by multiple applications. In this case, the actual execution times of service units of different applications must be independent. TDM is a simple arbiter that achieves this [18], but it is also possible to use any predictable arbiter (RR, CCSP, etc.) in combination with delay blocks [3]. A *delay block*, shown

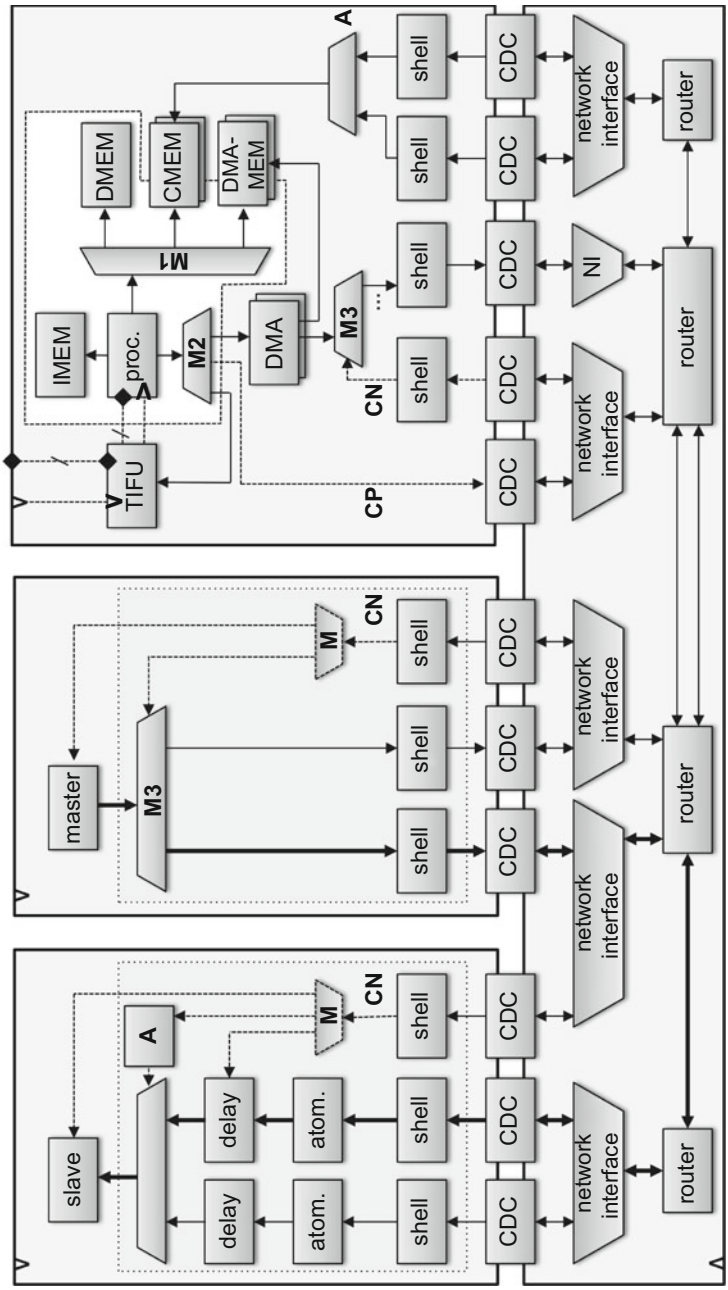


Fig. 16.2 Example CompSOC hardware platform instance. *Arrows* point from a master (initiator) port to a slave (target) port. *Dashed arrows* indicate control connections to program arbiters, memory maps, and IPBs. A is an arbiter, C is a control connection, and M are memory maps

in the slave tile of Fig. 16.2, only releases a response from the slave at the WCRT, cf. Fig. 16.1. Since the WCRT takes into account the worst-case interference from other applications, the delay block enforces a response time that is independent from other applications.

When arbitration has to be composable, predictable arbiters (RR, CCSP, etc.) with a delay block may offer more flexibility than a composable arbiter (TDM). In particular, since TDM inversely couples throughput and response time, a small WCRT is only possible with large budget, which the requestor may not need. Especially for memories that are loaded heavily, such as DRAM (see below), over-reservation may not be acceptable. On the other hand, TDM has the advantage that it is easier to reprogram the budget of a single application without affecting other running applications [18]. Resources that have small service units, such as memories, require arbitration to be implemented in hardware. It is then not practical to use a two-level arbiter that is composable between applications and predictable within an application, because it requires fixing the number of requestors per application in the hardware, which is expensive and inflexible.

The arbiter in the tile, the delay block, and the IPB may all be programmable, which is done by write transactions on the tile's slave port, which is then demultiplexed with a fixed memory map (M in Fig. 16.2) to the appropriate block.

16.3.2.2 SRAM

SRAM is the prototypical slave IPB. The service unit is usually as small as one word, which necessitates a fast (and thus simple) hardware arbiter. SRAMs are often shared within a single application, e.g., Communication Memory (CM), and round robin arbitration is then used. To share between applications, atomizers are required, and usually service units of a single word and a RR arbiter are used.

▶ Chapter 13, "Memory Architectures" gives more information about general SoC memory hierarchies.

16.3.2.3 DRAM

A DRAM tile has the same structure as an SRAM tile, but the DRAM itself has quite different characteristics. In particular, its service units are more complex [1, 4, 9, 17, 30]. Reading and writing in a DRAM require sending a number of commands (activate, read/write, precharge, refresh). For a reasonable efficiency, it is required to use bursts of data, typically with a length of eight words for most contemporary DDR memories, and then pipeline the DRAM commands to a single DRAM bank and/or across multiple DRAM banks. A traditional (non-real-time) DRAM controller schedules commands dynamically as service units arrive, using an open-page policy [4]. It is hard to analyze the execution time of each service unit because the time between successive DRAM commands varies a lot, depending whether the data that is accessed is in a bank that is open (activated) or not.

For this reason, CompSOC's real-time Raptor memory controller uses a close-page policy that ensures that the ET of a service unit is predictable [13] or even constant [18]. The memory commands for the service units have to be programmed into the memory controller, using the rightmost dotted line in the slave tile of Fig. 16.2.

With a predicable ET of a service unit, the execution time of a service unit can still depend on the preceding service unit, e.g., depending on whether it was a read or a write. This can be prevented by scheduling memory commands differently, resulting in composable service units that have a constant execution time. It has been shown in [18] that composable service units can be used with several generations of DRAM memories with a negligible impact on performance.

The DRAM is usually a heavily loaded resource and used by multiple applications. It is possible to use predictable service units with a predictable arbiter (e.g., RR, CCSP) and delay blocks (recall that the constant WCRT eliminates all interference) [3]. Alternatively, composable service units may be used, with a composable arbiter (TDM) [18], with the advantages and disadvantages discussed in the previous section.

16.3.3 Processor Tile

Having introduced basic master and slave tiles, we now discuss the processor tile, which is a mix of both. As shown on the right of Fig. 16.2, it contains a processor (in our prototype, a Microblaze or ARM Cortex M0) with an Instruction Memory (IMEM) and a Data Memory (DMEM). The memories are usually tightly coupled (i.e., have a single-cycle access time) using Instruction Local Memory Bus (ILMB) and Data Local Memory Bus (DLMB) busses.

There are no caches in the tile, because it must be possible to compute the WCET of a task on the processor if predictability is required. We do not use caches because the WCET would then not only depend on the processor but also on the time for cache misses. It is possible to take into account the WCET of the interconnect and remote (off-tile) memories to compute remote memory accesses [23]. However, we decided to keep the architecture and performance analysis simple and not use caches. A second, more important reason to omit caches is to adhere to Concept 1 of budgets: the WCRT of a task on a processor then only depends on the budget of the processor on which it runs. This allows the WCRT of all tasks to be computed independently of other tasks and of how their communication and storage are mapped on the platform. Performance analysis is thus compositional, i.e., consisting of independent smaller analyses. This simplifies the design and verification flow [14].

A processor cannot access remote memories directly. Otherwise, similarly to caches, the WCET of a task would depend on read and write transactions to a remote memory, i.e., the arbitration in the NoC and remote memory. Additionally, since sharing a processor between applications must be composable, its service units are enforced by preemption through interrupts, as explained in Sect. 16.4.1. However, since simpler processors, such as Microblaze and ARM Cortex M0/3/4, do not allow read and write transactions to be interrupted, the interrupt service latency could be very long. By using a DMA to access remote memories, the data transfer between local and remote memories is executed independently and concurrently with the

processor. The task that programmed the DMA can be interrupted in a few cycles and swapped out, for an efficient implementation of composable sharing of the processor.

The execution time of a task on the processor may depend on tasks that executed before it, due to the processor's internal state. For example, when the processor implements branch prediction, the predictor state is not automatically reset between task switches. Cache pollution is another example, although we already eliminated it by excluding the use of caches. Since composability requires that the execution time of an application is independent of other applications (running earlier or concurrently), we ensure that the processor is reset to a *neutral state* [14] between application switches by resetting the branch predictor state. For caches, it would be enough to flush them between task switches. In general, an instruction to reset the processor's entire internal state would make it easier to make it predictable and composable.

As already mentioned, and shown in Fig. 16.2, a processor uses DMAs to read or write data in a remote memory. Each DMA connects to a port of a DMA Memory (DMAMEM). The DMAMEM connects with another port to the processor DLMB (other options are discussed in [34]). Using two ports on the memory avoids arbitration between the processor and the DMA, which would necessitate changing the single-cycle turn around DLMB bus to a slower PLB bus. (When using an ARM processor, these would be AHB or AXI busses.) The DMA has a Memory-Mapped I/O (MMIO) port on the PLB, through which it is programmed to either copy data from the DMAMEM to a remote memory accessed over the NoC or vice versa. The DMA has a NoC connection to each remote memory. Since it is cheap, a DMA is not shared between applications to avoid interference. Although it may be used (sequentially) by different tasks of the same application, avoiding this simplifies the computation of the WCET of a task by eliminating interference from other tasks.

A remote memory can be the shared DRAM, a shared SRAM on the NoC, or an SRAM in another (remote) tile. The latter is a CM: the DMA can copy from the local DMAMEM into a CM of another processor tile, or vice versa. The CM has two ports, for the same reason the DMAMEM does. If multiple remote DMAs can access the same CM, a multiplexer with an arbiter is required (labeled A in Fig. 16.2, cf. Sect. 16.3.2).

Execution on the processor can be preempted with interrupts. This is required as soon as multiple applications run on the processor because applications should run in an interleaved fashion, not consecutively. The service unit of a processor is a TDM slot (or time slice), generally lasting 20,000 cycles or more. To achieve composability, tasks must be preempted at the end of each TDM slot, after exactly the same number of clock cycles. For this, a Timer, Interrupt, and Frequency Unit (TIFU) is attached to the processor. It can generate interrupts at precise moments in time and halt (clock gate) the processor until precise moments in time. The interrupt and clock lines from the TIFU to the processor are shown in Fig. 16.2 to and from the TIFU. The TIFU can also change the frequency at which the processor can run for power management [38], shown by the clock domain in white. Section 16.4.1 describes how the TIFU is used to preempt and arbitrate the applications and tasks within an application on a processor.

The processor tile is the most complex because the composable service units require preemption and decoupling the processor from other IPBs (in particular, remote memories). This requires the TIFU, multiple local memories, some with their own arbitration, and DMAs.

16.3.4 Network-On-Chip

The NoC allows all IPBs to communicate. ▸ Chapter 15, "Network-on-Chip Design" contains a thorough introduction to NoC technology. From a real-time perspective, the NoC is problematic because it is a resource that is made of smaller components, namely, routers and Network Interface (NI), that are distributed spatially. A single centralized arbiter is thus not feasible, and each router and NI has an independent local arbiter. As a result, a service unit that enters the network may be delayed at each router that takes local decisions based on its local state only. This makes it very hard to compute the end-to-end response time and throughput due to contention and congestion. The CompSOC platform uses the synchronous daelite NoC [49], which is derived from the mesochronous Aethereal NoC [22]. The asynchronous Argo NoC [27] is based on the latter. The common underlying concept is to use a *single global TDM arbiter for the entire NoC but to implement it in a distributed way*. The routers are synchronized such that service units (flits) never arrive at the same link at the same time, thus eliminating all contention. Without contention, routers require no arbiters and no buffers (except for pipelining), making them very fast and cheap [15]. Service units only wait at the edge of the NoC, in the NIs [45]. To offer real-time timing guarantees, the global TDM schedule is programmed at run time, to offer connections, i.e., resource budgets, from a master IPB initiating transactions to a slave IPB executing transactions.

As illustrated in Fig. 16.2, the slave, master, and processor tiles, and the NoC operate in independent clock domains, which are straddled by the Clock Domain Crossing (CDC) blocks. The CDCs are functionally transparent, and we will not discuss them further. The NoC thus implements the GALS paradigm, which is essential for scalability (Concept 4).

The four components [21] of the NoC are illustrated in Fig. 16.2, and we described them following the bold arrows from a master IPB to a slave IPB and back. First, transactions from a master IPB may go to multiple slaves according to its memory map, discussed in more detail in the next section. To implement the memory map, a programmable demultiplexer sends each request to the connection to the right memory and interleaves responses in the right order. Second, the protocol shell serializes each transaction to a stream of data words, without any particular structure. This allows multiple IPB protocols, such as AHB and AXI, to be sent over the same NoC. Third, NIs preempt the data stream into service units (flits) that are injected in the NoC according to the TDM schedule. Finally, routers just move flits around, without contention and hence without arbitration. When a flit arrives at its destination NI, the constituent words are given to the protocol shell that reconstitutes the transaction in the right protocol.

Intuitively, the real-time performance of the NoC, in particular the WCRT and throughput of a connection are easily computed by looking at the number of slots reserved for the connection and the total number of slots in the TDM table. However, an additional complication is that NI buffers are finite and slave IPBs do not necessarily accept incoming transactions immediately. To avoid dropping data, NIs use end-to-end flow control, which must be modeled in the WCRT calculation [24].

The service unit of the NoC is a flit, and the protocol shells together with the NIs preempt larger transactions into flits. The NoC uses single-level TDM arbitration between flits of the same as well as different applications to eliminate costly per-application or per-connection buffers in the router. Although two-level arbitration is possible, it makes routers larger and slower [15].

16.3.5 Peripherals

Many peripheral tiles are simple non-shared slave tiles. For example, a Universal Asynchronous Receiver/Transmitter (UART), Serial Peripheral Interface (SPI), or audio peripheral is usually not shared between multiple requestors. As a result, they can be hooked up directly to the NoC, with only a slave port, and delay blocks and arbitration are not required. A processor usually accesses the peripheral directly (using its DMA), usually polling for data or writing data into the peripheral's local buffer.

As an example of a complex peripheral, the inter-NoC bridge [33] links the NoC on the Field-Programmable Gate Array (FPGA)/Application-Specific Integrated Circuit (ASIC) to a NoC on another FPGA/ASIC or even to another computer. It acts as both slave and master. As a slave, it receives data from the local NoC over multiple connections to be sent to the remote NoC. As a master, it produces data on multiple connections that came from the remote NoC. Given TDM arbitration in the NoCs, the bridge uses TDM arbitration too. However, its service unit depends on the medium over which the NoCs are connected. We used an Ethernet Media Access Control (MAC) with Xilinx RocketIO with large Ethernet packets for high data efficiency, but at the cost of a large scheduling interval and WCRT. The inter-NoC bridge is programmable and thus also has MMIO slave ports.

16.3.6 Memory Map

All the components that we have described until now communicate with each other using transactions. A transaction can read or write in a memory at a given address or address range. Additionally, IPBs and their tiles can often be programmed by writing into memory-mapped register. Similarly, data input or output is often performed through memory-mapped buffers.

The CompSOC platform implements distributed shared memory. It is distributed in the sense that there are multiple memories in the same 32-bit memory map, and shared because multiple IPBs can access a given memory. The memory map

defines in which memory each of the 32-bit addresses is located. It is implemented in several places. First, hardwired demultiplexers. The demultiplexer (labeled M1 in Fig. 16.2) is the local DLMB bus in the processor tile. It is hardwired with the address ranges of the IMEM, DMEM, CM, and DMAMEM memories in the local tile. The demultiplexer labeled M2 is similarly hardwired and decodes the local PLB peripheral bus. Finally, the demultiplexers labeled M decode the MMIO transactions to program a slave and its arbiter and delay blocks.

Second, the programmable demultiplexers M3 and NoC connections define the remainder of the memory map. Given an address, the former first selects a NoC connection, and the latter then delivers the transaction to the memory (more generally, IPB), as illustrated by the thicker lines from the master IPB to the slave IPB. Note that the routers and NIs implement part of the memory map, but they have no knowledge of transactions or memory addresses: they just implement point-to-point connections to transport data (requests, as it happens) from master IPBs to slave IPBs and vice versa (responses). This simplifies the NoC, making it faster. It also enables multiple memory maps to coexist in the same platform. For example, multiple processors may boot from (their) address 0, which is mapped in different memories for different processors. This is useful when processors run different Real-Time Operating Systems (RTOSs) or are heterogeneous.

There are no NoC connections after a reset, and IPBs cannot communicate with each other at all. The memory map (multiplexers M3 and NoC connections) is programmed using the NoC itself at run time, using a bootstrap procedure [20,49]. A privileged processor sets up NoC connections one by one and programs the memory map(s), offering a secure boot procedure. It directly programs the NoC using the dotted control line CP in Fig. 16.2. Some memory-map multiplexers may be hardwired, e.g., M. All others (M3), including those of the DMAs of all processors, are programmed over the NoC using control lines CN [21]. Programming a memory map is predictable and composable.

16.3.7 Atomicity

The NoC allows master IPBs to communicate using distributed shared memory. When (a task on) an IPB sends data to a shared memory, it is important that it has been written completely before it is read by another IPB. However, high-performance communication protocols, such as AXI, limit the atomicity of transactions to a single byte. It is hence not possible to send more than a single byte atomically without further precautions.

The CompSOC platform addresses atomicity at several levels. First, the hardware reads and writes 32-bit words atomically to all data memories and memory-mapped IPBs. Second, recall that each resource has a minimum service unit that is executed atomically, i.e., without interruption or preemption by other service units (they may be pipelined though). Transactions that read or write more than a service unit, or are not aligned, are chopped into multiple service units that may be arbitrarily interleaved with service units of other requestors. The minimum service unit of

SRAM is a single 32-bit word, and that of a DDR3 DRAM typically eight words. Thus, an IPB can atomically access data in a memory, as long as it is no larger than one service unit.

Regarding atomicity of computation, the service unit of a processor that runs multiple tasks is a time slice (Sect. 16.4.1). The task that runs is interrupted and preempted (swapped out) for another task at the end of each slice. Since interrupts can occur after each instruction, only a single instruction is atomically executed from a software perspective. Traditionally, atomicity is extended to multiple instructions (called a critical region) by disabling the interrupt before the critical region and reenabling it at the end. However, critical regions increase the . In the worst case, a non-real-time application enters but never exits a critical region, so claiming a shared resource forever. To avoid this, we do not allow critical regions in application code. We will return to this topic in Sect. 16.4.1.

Although atomic transactions are essential, they are not enough to safely communicate data. Data may be (atomically) overwritten before it has been read, for example. This will be addressed in the synchronization Sect. 16.4.4.

16.3.8 No Synchronization Hardware

According to the scalability Concept 4, most MPSoCs are based on the GALS paradigm. Each IPB or tile operates synchronously on its own clock but is asynchronous with other IPB or tiles, as discussed in the NoC Sect. 16.3.4. The TIFU in each processor tile provides local timers, but they are not synchronized with each other and may drift arbitrarily. Since there is no global clock or notion of time, pure time-triggered synchronization (whereby the parties wait until a specific time) is not possible. All synchronization strategies used in CompSOC, later described in Sect. 16.4.4, are therefore based on data synchronization.

We implement data synchronization based only on atomicity of read and write transactions. We do not use processor instructions, such as a test-and-set, or load-linked and store-conditional, or dedicated hardware for locks or mutexes. Test-and-set (and similar) instructions work by locking the path from the processor to the memory from the read (test) to the write (set) instruction. This is not scalable and infeasible when the processor and memory are connected by a distributed and shared interconnect, such as a NoC. Load-linked and store-conditional instructions work without locking the interconnect but require a processor with support for them, as well as a memory that keeps track of changes to its memory locations, which is not standard.

Locks and mutexes also require dedicated hardware, which we do not (wish to) have. They have an additional problem, namely, that when a resource is shared between multiple applications, it should never be locked by a single application, because that violates composability. At best a lock per application can be used, making sure that the resource is aware which requestor belongs to which application. However, in our platform, the NoC, SRAM, and DRAM have simple and fast single-level arbiters that do not distinguish between different applications.

The NoC and memory maps allow only atomic communication of memory service units. In Sect. 16.4.4, we describe how CompSOC offers safe communication and synchronization in software.

16.3.9 Conclusions

CompSOC is an example of a platform that supports running multiple applications with different time criticalities. In this section, we described the basic components comprising the CompSOC platform and justified the design choices for those components in accordance with the high-level concepts of the previous section.

In particular, each shared resource (i.e., processor, NoC, SRAM, DRAM, some of the peripherals) offers a service unit that is atomically executed with a known WCET. Requests are chopped into, or rounded up to, one or more service units.

Next, the arbitration policy is essential. To share a resource between (requestors of) non-real-time applications only, any arbitration policy may be used. However, to share a resource with requestors of multiple real-time applications, the resource arbitration must be predictable. This makes it possible to compute the WCRT from the individual WCETs and the predictable arbitration policy. If a resource is shared between real-time and non-real-time applications, which may not have a finite WCET, then arbitration must at least be composable between applications, and be predictable for requestors of the same real-time application. This can be implemented with a single-level composable arbiter (e.g., NoC, DRAM) or with a two-level arbiter (composable between applications, predictable or not within each application) as done on the processor (explained in Sect. 16.4.1). Composable arbitration can be implemented with TDM (e.g., NoC, processor, DRAM) or with any predictable arbiter plus delay blocks (e.g., SRAM, DRAM). Many predictable arbiters are available, such as RR, TDM, CCSP, or FBSP. Which one is used depends on the characteristics of the resource, such as cost of preemption and buffering, and the real-time requirements of the requestors.

16.4 Software Architecture

The CompSOC hardware platform contains computation, communication, and storage resources. Almost all can be shared between multiple requestors, and almost all can be (re)programmed at run time. The CompSOC software extends the single hardware platform to offer multiple Virtual Execution Platforms (VEPs). A VEP is an execution platform that is a subset of the CompSOC hardware platform, in terms of time (e.g., time multiplexing a processor) or space (e.g., non-shared DMA or a region in a memory). Each application runs in its own VEP, which is created, loaded, started, and possibly stopped and deleted, at run time. A CompSOC platform can run multiple VEPs concurrently, without any interference between them, i.e., composably.

To implement VEPs, the CompSOC software is constructed in a number of layers, as shown in Fig. 16.3. From the lower to higher layers, we have:

1. A *microkernel or RTOS* to share a resource with a software arbiter. Software arbitration is only used for the processor, since the service unit of all other resources is too small to be managed in software.
2. A *driver* library is used to program, load, start, use, and stop a virtual resource.
3. The resource requirements of requestor are specified using a *budget*. Reserving part of a resource for a requestor according to its budget results in a *virtual resource*. Most resources may be programmed to be shared in time and/or space. A VEP is a hierarchical virtual resource, made up of, for example, a region of DRAM, some NoC connections, and several virtual processor tiles. A virtual processor tile is again a hierarchical virtual resource, consisting of several memory regions, DMAs, and a virtual processor. The *budget management* (BM) library is used to reserve and release virtual resources, and thus keeps track of who has been reserved what part of the resource (TDM slots, memory regions, etc.).
4. The resource driver and budget management libraries comprise the *resource management* library.
5. When an application runs within its VEP, its tasks will need to synchronize and communicate, according to some *model of computation*. We offer several programming models or models of computation (threads, dataflow, Kahn Process Network (KPN), time-triggered) that are implemented using more primitive communication (barriers, FIFOs, sampling).
6. A *bundle* contains the code and data of an application together with the specification of the VEP it requires to run.
7. The *system application* manages a CompSOC platform. It uses the resource management library to create and remove VEPs and uses the *bootloader* library to start and stop applications.
8. Finally, an application programmer creates *applications* using one of the supported programming models or model of computation. He or she develops and validates the application in a VEP, which is either given up front or codefined with the application. The application is deployed as a bundle, i.e., the application's code (ELFs) together with the definition of the VEP. Composability guarantees that executing an application (bundle) before and after integration with other applications (bundles) gives exactly the same results.

Since the system software extends the hardware, it has the same requirements, and it must adhere to the same requirements and concepts. We now discuss each of the software components.

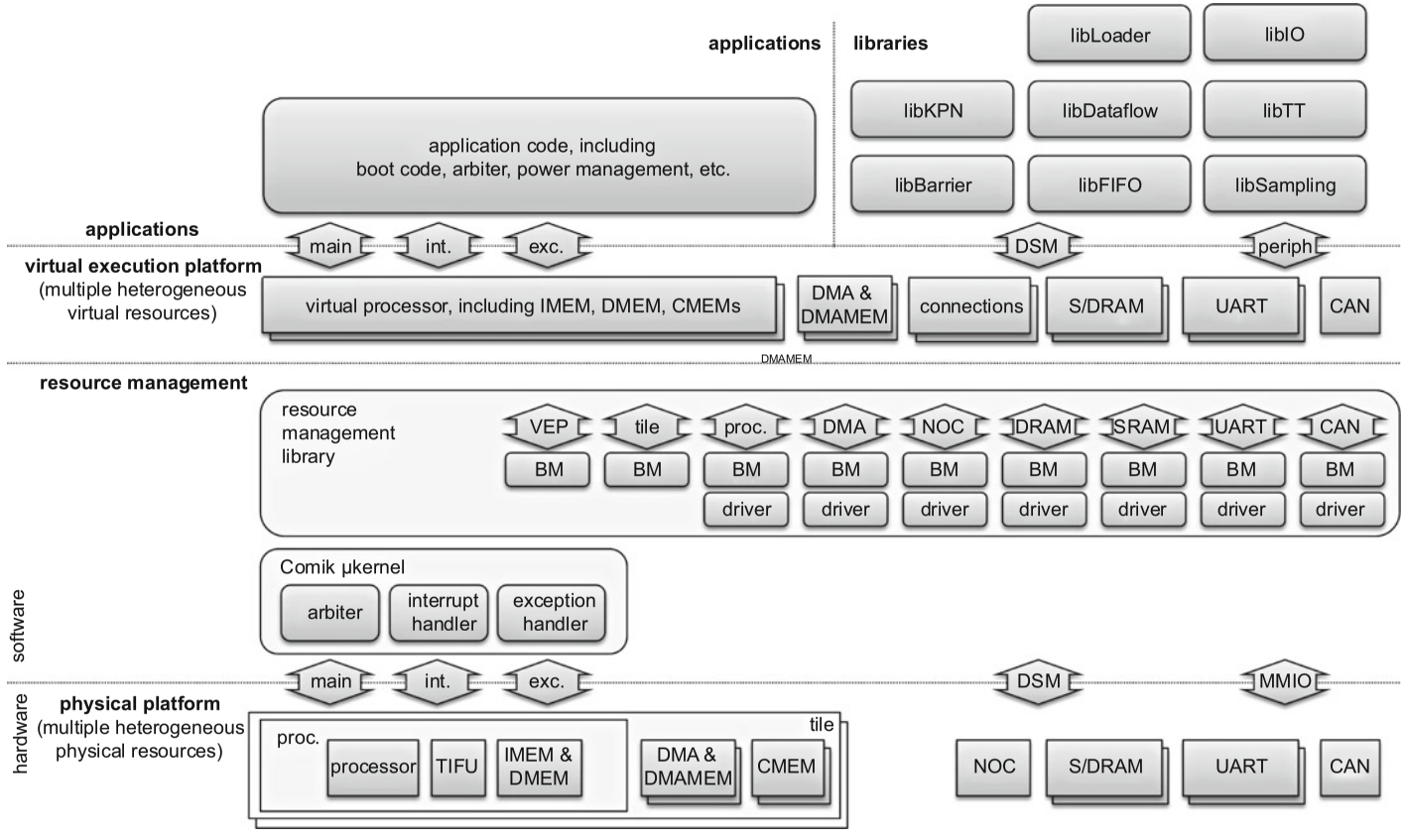


Fig. 16.3 Software architecture. DSM is distributed shared memory

16.4.1 Microkernel and RTOS

Processors are the only resources that are shared with software arbitration; all other resources use a (programmable) hardware arbiter. This is possible because the service unit of a processor, called a time slice, is in the order of tens of thousands of clock cycles and the cost of saving the state of the running application and arbitration is a few thousand clock cycles.

Recall that the concepts for a mixed-time-criticality platform include composable arbitration between applications (inter-application), predictable arbitration within an application (intra-application), or no arbitration for non-shared resources. Inter- and intra-application arbitration may be either separate (two-level) or combined (single-level). In the latter case, it must be both predictable and composable.

Task arbitration can be classified along several axes. First, it may be absent when there is only one task on a resource. Otherwise it is required. Second, it may be preemptive or not. Third, arbitration may be static and follow a static-order schedule or be dynamic where the order of tasks is determined at run time. Figure 16.4 and Table 16.1 illustrate how a processor can be (a.A) not shared or shared between tasks of one application only, either (a.B) with a static-order schedule, or (a.C) cooperatively scheduled (i.e., non-preemptive dynamic). Alternatively, multiple applications can share the processor using a microkernel such as CoMik, which arbitrates only between applications (b and c). Each application can use the techniques from (a) or even a virtualized RTOS, such as μC-OS III (c), to independently arbitrate between application tasks. The two levels of arbitration can also be combined in a single software arbitration layer (d), as done by the CompOSe RTOS [6, 19]. Finally (e), a traditional RTOS can be used with a single-level arbitration, e.g., TDM. However, if, as is often the case, priority-based arbitration without delay blocks is used, only a single application can use the processor.

As Fig. 16.4 shows, a physical processor can run software ("main") but also optionally has an interrupt handler ("int.") and exception handler ("exc."). When running the application directly on the processor (a), at least the main software must be defined by the programmer. A microkernel virtualizes the main, interrupt and exception handling and passes them on to the application (b) or RTOS (c). An RTOS that runs directly on the processor typically hides the interrupt and exception handling from the applications. In almost all cases, the programming model or model of computation used in an application hides interrupt and exception handling (see Sect. 16.4.4).

Because the execution of an application should not depend on another application, inter-application arbitration must be preemptive with service units (time slices) that are of constant length (down to the level of a clock cycle). As mentioned in Sect. 16.3.3, the TIFU allows the microkernel or RTOS to interrupt the running application at a precise point in the future. After saving the application context, the next application is scheduled, and its context restored. The time from the interrupt to the restored context may vary, e.g., due to critical regions, multi-cycle instructions, or variable execution time. Critical for composability CoMik and

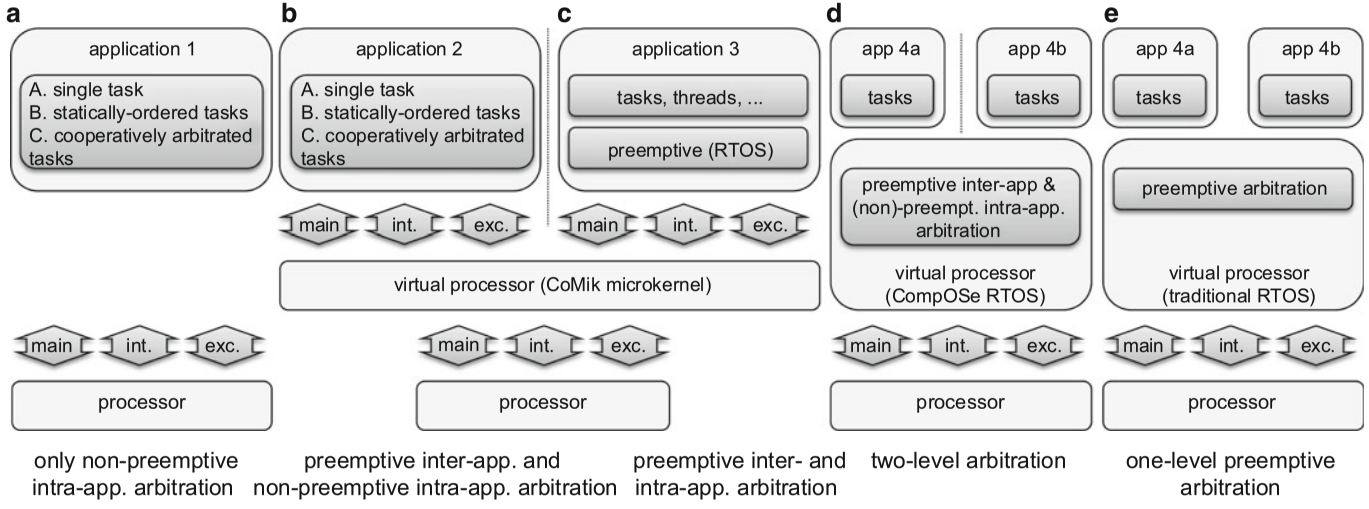


Fig. 16.4 Processor virtualization

Table 16.1 Inter- and intra-application arbitration on processors and other resources

	Inter-app.	Intra-application		
Figure 16.4	Preemptive	Preemptive	Non-preempt	Scenario
(a.A)	–	–	–	Single task
(a.B)	–	–	Static order	Dataflow actors of 1 app.
(a.C)	–	–	Cooperative	Tasks & FIFO, KPN processes of 1 app.
(b.A)	TDM	–	–	CoMik microkernel, multiple apps.
(b.B)	TDM	–	Static order	CoMik microkernel, multiple apps.
(b.C)	TDM	–	Cooperative	CoMik microkernel, multiple apps.
(c)	TDM	RTOS	–	CoMik microkernel, multi-app. incl. RTOS
(d)	TDM	–	Any/pred.	CompOSe RTOS with 2-level arb.
(d)	TDM	Any/pred.	–	CompOSe RTOS with 2-level arb.
(e)	–	Priority	–	Traditional RTOS with 1-level arb.
(e)	TDM		–	CoMik or traditional RTOS with 1-level arb.
–	–	–	Any/pred.	CM, 1-level arb.
–	TDM & composable		–	NoC, DRAM, SRAM, 1-level arb.
–	Atomiser & composable		–	DRAM, SRAM, 1-level arb.

CompOSe uniquely achieve application slots of constant length (at the level of an individual cycle) by halting (clock gating) the processor until the WCET of the CoMik slot [19, 35].

16.4.2 Drivers

Drivers are software libraries that shield programmers from hardware-specific details. More specifically, we use drivers to:

1. Program the (virtual) resource before use (e.g., set up a NoC connection).
2. Use the virtual resource (e.g., program a DMA transfer, change the frequency of a processor).
3. Reset the virtual resource after use (e.g., remove reserved TDM slots).

We discuss each of these briefly below.

Hardware resources often require that they are programmed before they can be used. For example, an arbiter in front of a shared resource (e.g., SRAM or DRAM) needs to be aware of the budget of each requestor. For TDM, this means programming the slots reserved for each requestor, and for priority-based arbiters like CCSP or FBSP, a static priority is required. Delay blocks must be programmed with parameters to dynamically compute the WCRT, if they are used. If a resource is reprogrammed while it is active, e.g., to add or remove a requestor, then often extra care must be taken to not invalidate predictability or composability of requestors that are running [18]. To release a requestor's budget is usually harder than to add

it, because we must ensure that no service units are still in the resource pipeline. We will return to this issue in Sect. 16.4.3. It is the task of a driver to ensure that a requestor's budget is programmed and reset correctly.

Furthermore, hardware resources often require that they are programmed in a certain way for use. For example, to send a block of data using a DMA, it must be programmed with the start address of where the data resides in the source memory, the block size, and the start address in the target memory. How to use the DMA also depends on whether it is used in a blocking or non-blocking manner. A driver takes care of these details and ensures that a DMA is used correctly.

The driver of the TIFU allows applications to access (virtualized) timers, schedule application interrupts, place the processor in sleep mode, and set the processor frequency. The driver ensures that these actions are done composably, i.e., without affecting other applications.

Two of the three types of driver functions (program, reset) are privileged in the sense that they manipulate the requestor budget on the resource. For this reason, their use is limited to the system application. The remaining driver function to use the resource is available to both the system and user applications.

16.4.2.1 Example Resource Drivers

The SRAM controller does not require programming, but its arbiter may do. It depends on the arbiter what has to be programmed, as discussed above. In fact, since the same arbiters can be used in front of the SRAM, DRAM, and similar resources, we use a general "shared resource" arbiter for all of these. Unlike the SRAM, the DRAM controller must be programmed with the DRAM commands that define the service unit [13], as well as parameters for logical to physical address translation, and this is done by the DRAM driver.

The NoC driver is complex because the source and destination NIs of a NoC connection must be programmed from the NI to which the driver is connected. The driver programs a path, TDM slots, flow-control credits, and enable/disable registers for each connection [22, 45, 49]. However, the NoC connection can be used without a driver after it has been programmed.

Before a requestor (application) can run on the processor, the microkernel must be programmed. This is straightforward since it uses a TDM arbiter that is predictable and composable. The processor is the only resource with frequency-scaling and clock-gating capabilities, for which the TIFU is used. Applications are allowed to change the frequency of the processor, while they run. However, as soon as their time slot ends, the frequency is first reset to the maximum frequency for the microkernel, and when it finishes, the frequency is reset to that of the next application [36]. This procedure is required to enable applications to manage the frequency at which they run without affecting the timing behavior of other applications, i.e., in a composable manner. Similarly, each application can use virtualized TIFU timers to measure time, to sleep for a certain time, to wake up at a certain deadline, and so on.

16.4.3 Virtual Resources and Their Management

Recall that each application runs inside its own composable VEP. A VEP is a hierarchical virtual resource that includes virtual DRAM, NoC connections, and virtual tiles further subdivided in virtual processors, DMAs and DMAMEMs, CMs, etc. Each virtual resource is the result of the driver programming a budget in a resource. The (hierarchical) budget is therefore the specification of the capacity that a requestor receives on a (hierarchical) resource.

Budgets are used to keep track of the reservations on the resource, such that the resource is not overloaded and such that multiple requestors do not use the same memory locations, time slots, etc. Without budgets, it is not possible to guarantee a minimum service to each requestor; in other words, using the resource would not be predictable.

The budget manager [16] is a library that offers several functions that are illustrated in Fig. 16.5. First, given a budget, the *reserve* function checks if the requested capacity (slots, memory range, energy, etc.) is available on the resource. If not, then the reservation fails. Therefore, loading a bundle, i.e., the application code plus its hierarchical budget, fails when not all of its requested virtual resources

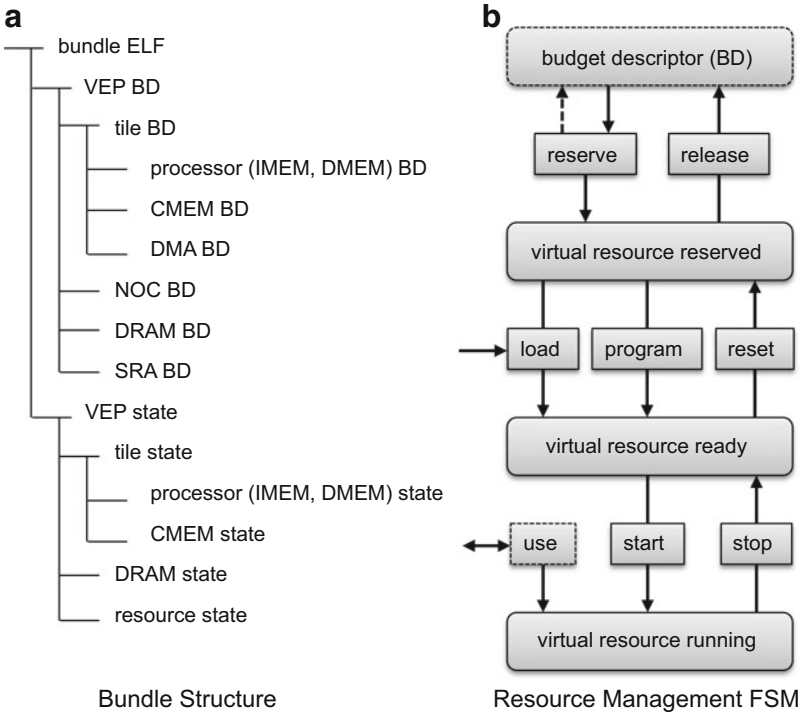


Fig. 16.5 (**a**) Bundle structure with Budget Descriptor (BD). (**b**) Resource management state diagram

are available. This is desired, since it is not allowed to modify the VEPs of other running applications, as that would not be composable.

A successful budget reservation results in a *budget identifier* (cf. file handle), with which the budget can be *programmed* in the resource by the driver. Resources often have to be *loaded* with an initial state before they are started. For example, a virtual processor's IMEM requires a main function, interrupt and exception handlers, and its DMEM may need to be loaded with initial data. A DMA (driver) has to be loaded with a memory map to be able to perform range checking for memory protection. Programming defines the virtual resource, while loading defines the state of the virtual resource.

It is important to decouple reserving, programming, and loading. Since programming and loading a resource can be slow, and applications always require a set of virtual resources, it is better to first try to reserve all of them, before programming and loading them. Otherwise, the programmed resources have to reset. When all virtual resources specified by a hierarchical budget have been programmed, i.e., the VEP is ready to run, it can be *started*. As resources are programmed sequentially, and even from different processors, all virtual resources should be created before the application starts running. Otherwise, problems may occur, such as a DMA sending data before its NoC connection has been created. At this point in time, whatever software has been loaded in the VEP will run with the resource performance that was specified in the budget.

Applications *use* a running resource, either using a driver (e.g., for the DMA) or without (e.g., for the NoC and S/DRAM). Some resources, such as the processor and its TIFU, may be programmed (in a safe way) by the application. For example, the processor frequency may be changed by an application, and it is possible to read timers and program interrupts at deadlines.

To stop an application in its VEP and release its (hierarchical) budget, the reverse process is followed. First, all virtual resources must be *stopped*. This may be intricate because resources may be pipelined, requests may be waiting in buffers before the resource, and requests may flow through several resources. For example, a software task may write to DRAM using DMA and NoC. In general, software tasks on the processors are stopped first. For the mentioned example, the pipeline of DMA, NoC, and DRAM will be empty after some time. (For a real-time application, this time is known). It is important that *quiescence* of resources (i.e., being in the ready state) can be observed to avoid leaving a resource in an inconsistent state or to lose data. Then the DMA, NoC, and DRAM virtual resources can be stopped.

When all virtual resources are in the ready state, the virtual resource can be *reset*, and the budgets *released*.

The budget management library can only be used by the system application to ensure that applications cannot change their own VEPs, which is crucial for composability and predictability. Together, the resource driver and budget manager comprise the *resource management* library. Usually all resources of the same type are controlled from one location, the exception being the processor tiles that are managed locally. Resource management may thus be distributed in the platform (see Sect. 16.4.5 and [48]).

16.4.4 Synchronization Libraries and Programming Models

Tasks in the same application synchronize and communicate data. As described in Sect. 16.3.7, the CompSOC platform offers atomicity of data transfers up to a certain size, which is the minimum requirement to safely communicate. However, it is not enough, because atomically written data may be overwritten before it has been read. Fundamentally, there are two ways to synchronize, either based on data or on time. First, we describe the data synchronization styles offered by the CompSOC platform: barriers and First-In First-Out (FIFO) queues. After that, we describe how we offer timed synchronization in a synchronous platform or on top of barrier synchronization when GALS is used.

16.4.4.1 Barrier Synchronization

Barrier synchronization is implemented as a library. When using a barrier to synchronize, each task increments its own dedicated counter (starting at zero) and then waits until all other tasks have done so by repeatedly reading (polling) all counters. This is shown in Fig. 16.6a. Starting at barrier b, task 1 updates its counter to $b + 1$ and then polls task 2's counter. As soon as task 2 has updated its counter, barrier $b + 1$ is reached, which is observed by the next polls of both tasks. Each counter is a single word in a memory, and is written atomically and independently of all other counters. It does not matter if all counters together are read atomically or not because counters are only incremented, and it is therefore not possible to miss a barrier. However, the assumption that all counters start at zero is problematic.

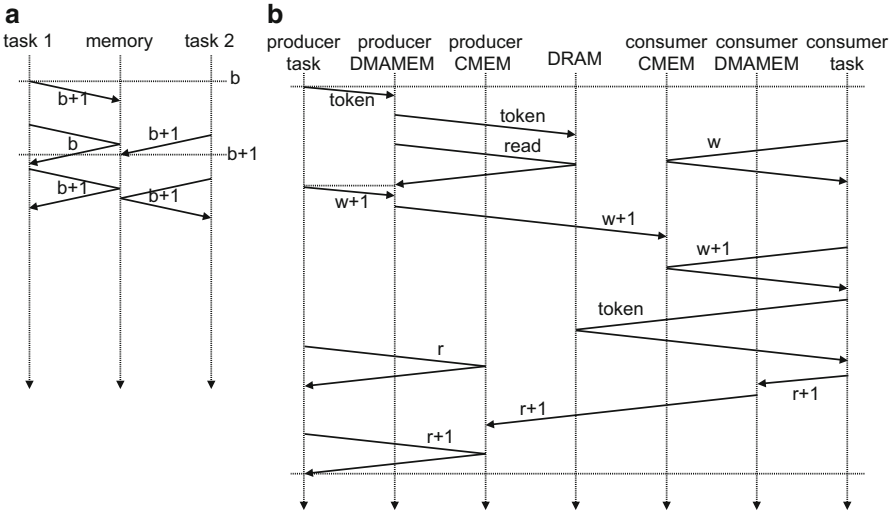


Fig. 16.6 (**a**) Barrier synchronization and (**b**) FIFO synchronization using DRAM. Each *arrow* points from master to slave (for the read/write request), possibly continued back to the master (for the read response). Labels indicate (b)arrier, (w)rite, and (r)ead data

At reset, SRAM and DRAM contain random values, and one task has to initialize the barrier counters before the others start, which is what the barrier intended to solve. For this reason all processors except the boot processor are kept in reset until their boot programs have been loaded and the barrier initialized.

16.4.4.2 FIFO Queues

In a streaming or data-driven system, tasks usually communicate using First-In First-Out (FIFO) queues that ensure no data is lost. Tasks block on a full or empty FIFO and wait until there is space to write or data to read. (In a cooperatively arbitrated system, a task could yield upon blocking.) Optionally, a task can poll for space or data, and continue with processing if it is not available. Although barriers work well to synchronize the flows of control of concurrent tasks or updates to global shared state, they are cumbersome for FIFO communication. For this reason, we offer a FIFO library that also forms the basis of KPN and dataflow libraries.

The C-HEAP protocol [41] implements FIFO in distributed shared memory without hardware support, such as locks, mutexes, or processor instructions (other than read and write). A FIFO has a single producer and consumer. Data is produced and consumed as constant-size tokens, and the FIFO has a fixed token capacity. We must ensure that tokens are only written by the producer in memory locations that are guaranteed to have been freed by the consumer (i.e., it has read the token). Similarly, the consumer must only read memory locations when the producer has finished writing the token, i.e., the data is valid. The basic concept is that the start location of valid tokens is indicated by *a write pointer that is only written by the producer* and that the start of space for new tokens is indicated by *a read pointer that is only written by the consumer*. The pointers are only updated after tokens have been written to or read from the destination memory. Because each pointer is only written (atomically) by one task, consistency of the FIFO is guaranteed. No token is overwritten before it is read or accidentally read multiple times. The difference between the pointers indicates the FIFO filling, and the FIFO is full or empty when the read and write pointer are equal. We use a circular buffer, which means that wrapping of pointers must be taken into account.

Figure 16.7a illustrates the simplest FIFO (L), where the tokens and the read and write pointers all reside in DMEM. Only tasks on this processor can communicate using this FIFO, as shown in Fig. 16.7b. When a FIFO is too large to fit in the local memories of a processor tile, it can be stored in an SRAM or DRAM that is connected to the NoC, as shown by FIFO F in Fig. 16.7a. The protocol is unchanged, but multiple memories and DMAs are now used, as shown in Fig. 16.6b.

The producer of a token first checks for space for a new token (blocking *claim_space* or non-blocking *poll_space*). The token is then written into its local DMAMEM and copied (*write_token*) to the remote memory when it is ready. To ensure that the token has been written in the remote memory, the processor issues a (dummy) read of the last data word of the token, again using the DMA. Since the NoC connections do not reorder read and write transactions, the data is guaranteed to be written when the read data is returned. When the dummy read data has been

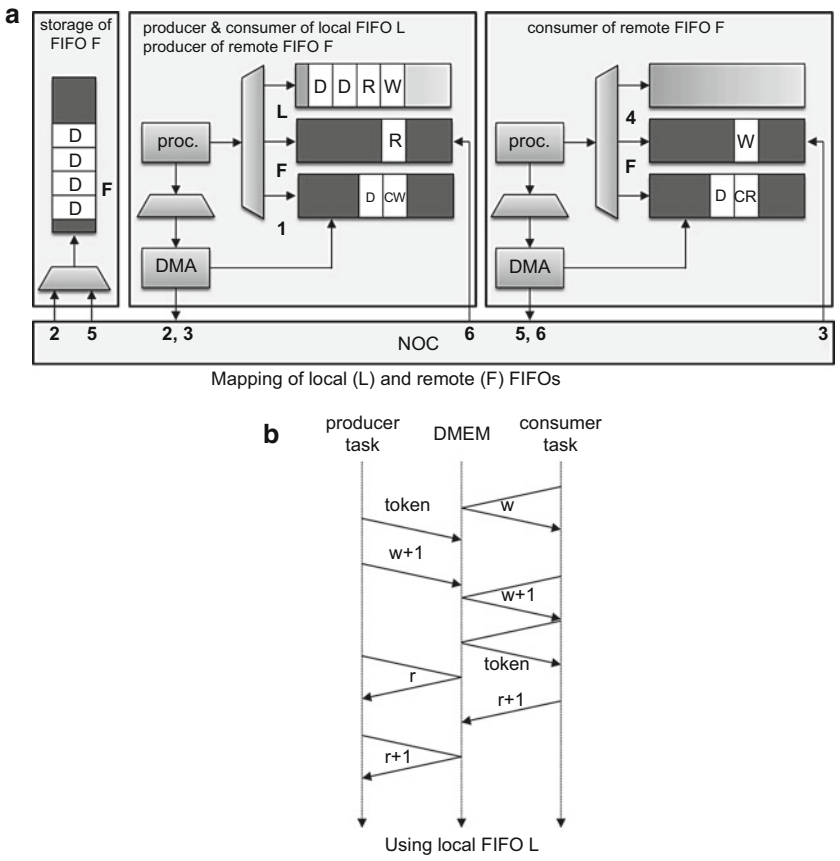


Fig. 16.7 (**a**) FIFO F with data in remote DRAM and FIFO L with data in local DMEM. (**b**) Using local FIFO L. *D* are tokens, *R* and *W* are read and write pointers, respectively

received and discarded, the write counter (CW) is updated ($w+1$) in the local DMAMEM, and the DMA is programmed to copy it to the CM (W) of the consumer (*release_token*).

Assume that the FIFO was empty until the write counter update. The consumer would have known this because the read CR and write W pointers in its DMAMEM and CM would have been equal: $w = r$ in Fig. 16.6b. The consumer would have been waiting until the write counter update (blocking *claim_token* or non-blocking *poll_token*). When the write counter update is detected, the consumer instructs its DMA to copy the token from the remote memory to its local DMAMEM (*read_token*). When the consumer no longer requires the token, it is released (*release_space*) by increasing the local read counter CR and writing it to the producer's CM (R).

The tokens and counters may be placed in any memory but preferably as close as possible to the task that requires reading most often. The highest performance

is obtained by maximizing the use of posted (non-blocking) write transactions and minimizing the number of read transactions over the NoC. This is achieved when placing the write counter W in the consumer's CM and the read counter R in the producer's CM, as well as keeping local copies of the counters (CW, CR). The data is best placed in the consumer's CM, assuming it fits. The *claim_space*, *write_token*, and *release_token* are often combined in a simpler *send_token*. *receive_token* is defined conversely. Although simpler, they require space for one more token and copy action.

16.4.4.3 Timed Communication

In a time-triggered system, jobs communicate through shared locations (memory regions) with space for one token. The token may be written multiple times before it is read, and read multiple times before it is written again. If under- and oversampling are not desired, then the usual way to synchronize the producer and consumer is by (statically) scheduling access to the shared location in time. For example, with token production with a period of 1 ms starting at time 0, we can schedule token consumption with the same period of 1 ms but with a starting time of 1 ms. As long as the producer ensures that the token has been written completely in the shared location before the period deadline, the consumer can safely read the token. This approach works as long as a sufficiently accurate common notion of time is available to the producer and the consumer, and the production and transport of the token is predictable and guaranteed to fit within the production and consumption periods.

In a fully synchronous CompSOC platform, i.e., where the NoC and all processor and memory tiles operate on the same clock, time-triggered communication works without surprises. The simplest scenario is that the producer computes the token and writes it to the shared location. Using the TIFU, it then sleeps until the periodic deadline and then restarts. The consumer(s) sleep until the periodic deadline and then copy the token [55]. Care must be taken to not write tokens too late (i.e., arriving after the deadline) and not too early (i.e., when consumers are still reading the previous token). Even without a global clock or notion of time, we can use barriers to safely implement time-triggered communication in a GALS MPSoC. We show a basic implementation in Fig. 16.8. A producer writes its token and increments and waits on the producer barrier. The producer barrier is released periodically by a clock task, after which consumers indicate in a consumer barrier when they have consumed the token. The producer waits for the consumer barrier before it updates the token and the producer barrier. Producers and consumers follow both barriers to detect any late consumption or production, which could result in transfer of corrupt data.

16.4.4.4 Programming Models

Different applications of different time criticalities will be developed with different requirements on timing. Real-time applications must be analyzable, which imply a more restrictive programming model (e.g., dataflow or time triggered) than a programming model that non-real-time applications can use, such as Kahn Process Network (KPN) or even arbitrary C. CompSOC therefore offers multiple

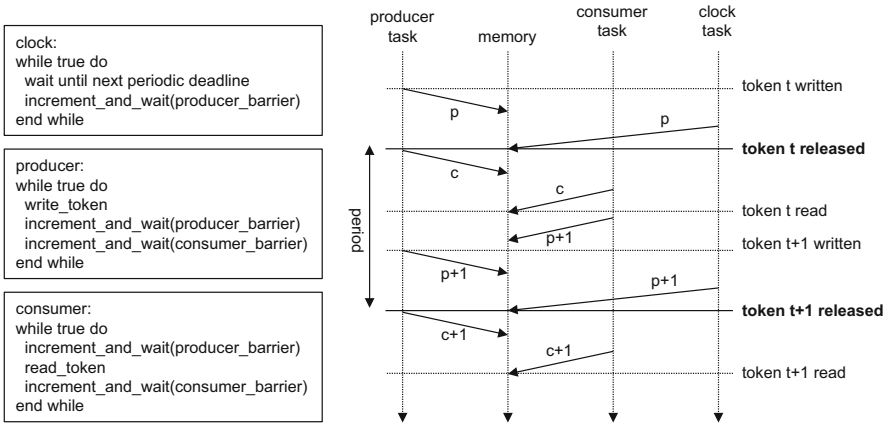


Fig. 16.8 Timed communication using barriers

programming models, such that each application can be designed as easily and efficiently as possible.

Although we do not encourage it, an application programmer can use multiple tasks and communicate using DMAs, without using CompSOC's synchronization libraries. All the programming models are implemented using tasks (or threads), using the templates shown in Fig. 16.9. We describe them from least restrictive, and thus least analyzable, to most restrictive [32]. In all cases, the task graph is static.

1. *Non-blocking and blocking FIFO communication*. The first arrow illustrates that *claim_space* and its corresponding *write_token* and *release_token* may have arbitrary code between them. The second arrow illustrates that this task uses polling, which could result in nondeterministic behavior. This is because it depends on the scheduling of other tasks, which may be non-deterministic due to GALS. This is not recommended unless the task uses some kind of sampling algorithm.
2. *Kahn Process Network (KPN)* with finite FIFOs use only blocking *send_token* and *receive_token*. Processes may use arbitrary control flow, which may result in a data-dependent number of tokens being consumed or produced (indicated by the arrow). It is, in general, not possible to compute the WCET of a process, and KPN is suitable for non-real-time or perhaps soft-real-time applications.
3. (Not shown in the figure.) Programmers can use *barriers*, as described earlier in this section, to synchronize multiple tasks using patterns such as fork-join.
4. *Dataflow* applications contain actors that consume all input tokens before computing the output tokens, which are released at the end. Note that an actor is a stateless function, unlike a KPN process. State can be implemented with a self-edge, i.e., a channel from the actor to itself. A programmer only writes the actor functions; the `dataflow_actor` wrapper is automatically generated. If the code in the actor function has a WCET, and all resources are shared

Fig. 16.9 Programming models

predictably, then the throughput and WCRT of the dataflow application as a whole can be computed [37]. As a result the dataflow model of computation is suitable for real-time applications. ▶ Chapter 3, "SysteMoC: A Data-Flow Programming Language for Codesign" contains more information about the dataflow model of computation, and ▶ Chap. 4, "ForSyDe: System Design Using a Functional Language and Models of Computation" contains more information about dataflow and other models of computation.

5. *Time-triggered communication*. A job (non-blocking) reads tokens from its input locations (except for the first execution), computes outputs, and then (non-blocking) writes tokens in its output locations. Jobs communicate using the time-triggered synchronization library. Time-triggered applications are executed periodically according to a global static schedule, for example, using a global clock or barriers. The code of each job (computation on processor, as well as communication on NoC, and access to shared memories) must have a WCRT smaller than the period, which determines the real-time performance of the application. As a result, the time-triggered model of computation is suitable for real-time applications. ▶ Chapter 24, "Networked Real-Time Embedded Systems" contains more information about the time-triggered model of computation.

16.4.5 System Application and Application Loading

The system application is like a normal application with the exception that it has access to privileged resource management library functions. The system application creates and removes VEPs, and starts and stops applications in running VEPs. The system application has a task on each processor tile, with one being the master. The tasks synchronize and communicate using the barrier and FIFO libraries.

Figure 16.10 illustrates the six (simplified) steps to start an application [48]:

1. The system application detects a bundle.
2. It hierarchically creates a VEP in a distributed manner.
3. It loads the bootloader in the VEP, and starts the VEP and bootloader.
4. In the VEP, the bootloader loads the application code and data.
5. For multitask applications, the bootloader creates tasks and communication channels.
6. The bootloader finishes and the application runs.

These steps are now explained in more detail. The master task detects in bundles at a predefined location in shared memory and loads them based on some strategy (e.g., as soon as they have been uploaded from outside the ASIC/FPGA or based on certain triggers [26]). The bundle contains the budget description of the VEP that the application requires to run.

The master task analyzes the hierarchical budget descriptor and sends the budget descriptors of the processor tiles to the slave tasks on those tiles. The slave tasks locally reserve and program the required resources and notify the master task. Other

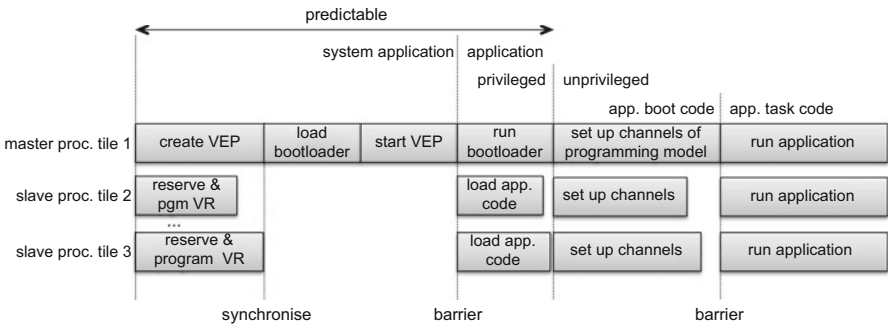


Fig. 16.10 Loading and starting an application

resources, such as the NoC and shared memories, are similarly reserved by the master or a slave task. If any budget reservation fails, the entire VEP creation fails, and all its resources are released. After the VEP has been created, it is loaded with a standard small bootloader application. After starting all the virtual resources, the bootloader loads and starts the application code that is specified in the bundle. In this way, loading an application, which may take some time, is performed in the VEP of the application instead of that of the system application. Multiple applications may be loading and/or running simultaneously because each VEP is independent.

The bootloader finishes and hands over to the application code. A non-preemptive application on a single processor (Fig. 16.4a, 16.4b) now runs. Otherwise, the application contains tasks (actors, processes, etc.) that synchronize or communicate using barriers, FIFOs, etc., and then these are set up using the relevant programming model library (see Fig. 16.3). When all tiles on which the application runs have finished this set-up phase, they synchronize using a barrier, and the distributed application starts.

The time from observing arrival of a bundle and starting the application is predictable and can also be made composable. In other words, the time from detecting a bundle until it is running is independent of other applications [48].

16.4.6 Conclusions

The software architecture of the CompSOC platform is quite complex, but this is mostly due to its versatility. The processor is the only software-arbitrated resource, and virtualization of multiple applications is inherently quite involved. The essential concept of the software architecture is the bundle which contains application code together with the requirements for its VEP structuring. Building on this, the resource management library provides a uniform way to reserve, program, and start heterogeneous resources in a hierarchical and distributed manner. After the system application creates a VEP, the bootloader library is used to load and start

applications, of any kind. Various communication and programming model libraries are provided for both system application and user applications.

16.5 Example CompSOC Platform Instance

The CompSOC platform is a template that can be instantiated [14] and used in many different ways and applications. In this section, we describe a demonstrator showing:

1. Mixed-time-criticality: several concurrently executing applications with and without real-time requirements.
2. Predictability: guaranteed performance for real-time applications.
3. Composability: multiple applications loading and executing composably.
4. Multiple models of computation.

The demonstrator, shown in Fig. 16.11, contains five applications:

1. A *high-performance real-time motion controller for a two-mass spring motion system* written as a time-triggered application, shown in Fig. 16.12. The controller is a software task that regulates the current to a DC motor that drives and

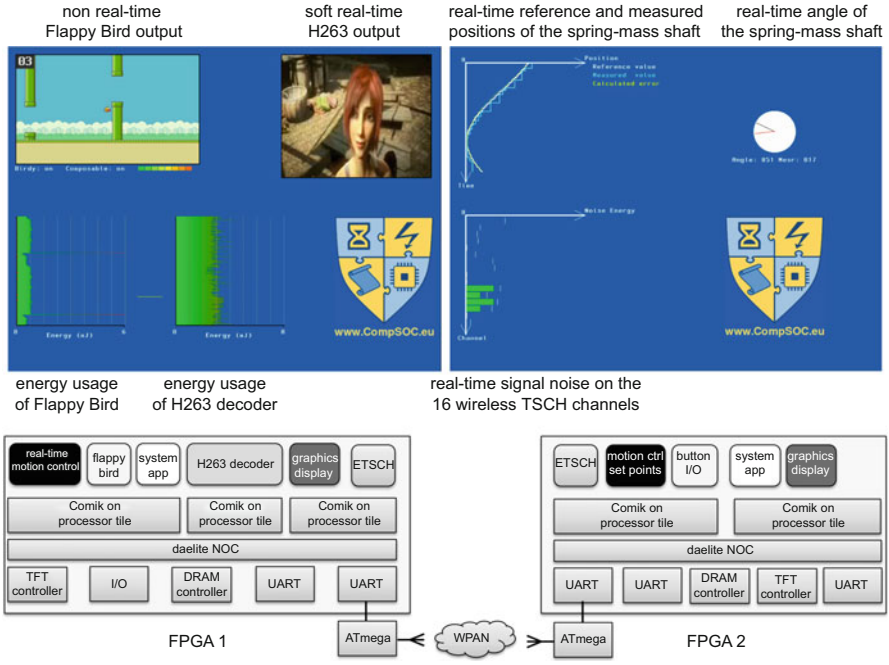


Fig. 16.11 Mapping of the applications on the two FPGA boards (*bottom*) and the display output of the two boards (*top*)

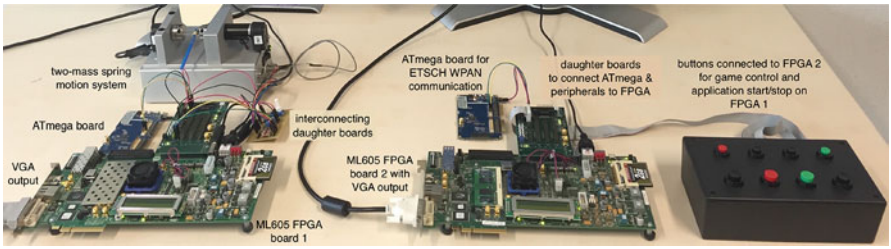


Fig. 16.12 Demonstration CompSOC hardware setup, showing two Xilinx ML605 FPGA boards with ATmega 256RFR2 boards for WPAN, interconnecting daughter boards, a two-mass spring motion system, and a video game controller

controls the rotational position of a flexible shaft [8]. The positions of the shaft are measured by optical encoders and read out by the software controller.

2. A CompSOC implementation of the non-real-time *popular video game Flappy Bird*. The user controls it with a simple hardware button. For demonstration purposes, the game can be played in composable mode, without interference from other applications, or in non-composable mode where the height of the bird's jumps depends on the processor load.
3. A *(soft) real-time H263 video decoder* [39], written as a dataflow application of six actors.
4. A *non-real-time graphics application* displays information on a Thin-Film Transistor (TFT) screen by updating a (triple) video buffer, which is read out by a hardware TFT controller. The graphics application has a (composable) sampling interface with the motion controller, Flappy Bird, and H263 video decoder to draw graphs and compose the image of several sub-images (video and game outputs).
5. The *system application* that manages the virtual execution platforms, including the starting and stopping of applications at run time.

   The applications are mapped on two Xilinx ML605 FPGA boards, shown in Fig. 16.12. The first board contains a CompSOC platform with three processor tiles each running the CoMik microkernel, NoC, DRAM, and several peripherals (TFT, UART) connected to the NoC. The sensing and actuation of the spring motion system is memory mapped on the NoC using an intermediate hardware I/O block. The second FPGA board has only two processor tiles and has the hardware controller for the Flappy Bird game instead of the motion-control hardware. Each ML605 board contains an Extended TSCH (ETSCH) application that connects to an ATmega 256RFR2 board using a UART. In this way, tasks of an application mapped on both FPGA boards can communicate wirelessly using the Extended TSCH (ETSCH) [51] extension of the IEEE 802.15.4e Time-Synchronised Channel Hopping (TSCH) standard. (E)TSCH uses frequency hopping and TDM for robust real-time performance. The architecture is GALS both within and between FPGAs.

Figure 16.11 illustrates the mapping of applications on (multiple) boards and (multiple) processors, as well as the display output of the boards. Of particular note are the following. Flappy Bird runs and displays on FPGA 1, but the game controller hardware (buttons) is connected to FPGA 2. The latency from the button I/O application on FPGA 2 to the Flappy Bird application on FPGA 1 is mostly due to the wireless connection and is just within the limits for a playable game. Using other buttons connected to FPGA 2, the system application on FPGA 1 can be instructed to start and stop Flappy Bird in different modes, without affecting other running applications. The graphics application on FPGA 2 displays the performance of the real-time motion controller with real-time graphs of the reference, measured angles of the spring-mass shaft and the calculated error. It also displays the real-time signal noise on the 16 wireless TSCH channels that it obtains from the ATmega board.

All the concepts of Sect. 16.2 are proven in the demonstrator. The system application dynamically loads application bundles by creating virtual execution platforms that are predictable and composable. Although the platform contains only five processor tiles in total, they are interconnected globally asynchronously using two TDM NoCs and a robust wireless TDM connection. The finite scheduling interval and efficient arbitration are optimizations and proven on the various resources, as described in previous sections.

16.6 Related Work

A vast literature is available on predictability, ranging from single resources to multiple shared resources, using a variety of analytical approaches, such as real-time calculus [52], dataflow [50], and response-time analysis for priority-based scheduling [10].

In the literature, composability is especially addressed for safety-critical applications, such as those found in the automotive [43] and aeronautical [46,56] industries. Temporal and spatial partitioning [46, 56] are addressed most often, increasingly using microkernels and RTOS, such as LynxOS-178, VxWorks 653, INTEGRITY, and PikeOS.

Our focus is on complete platforms that are predictable, are composable, or offer mixed-time-criticality. Note however, that in the literature, definitions of predictability and composability vary. Apart from CompSOC, notable mature platforms include the Transport-Triggered Architecture (TTA) [29], Giotto [25] and LET [28], and PRET [11]. A number of collaborative projects have developed platforms of varying degrees of maturity, including Flextiles [26], T-CREST [47], PARMERASA [54], MULTIPARTES [53], P-SOCRATES [44], DREAMS [42], and CERTAINTY [12]. The resource-management frameworks of [7, 31, 40] are noteworthy because their approaches consider the entire system, with different resources.

These platforms focus on different aspects, and they are often updated, which means that comparisons change over time. At a high level, all platforms offer

real-time performance to at least one application and sometimes to multiple applications. Dynamic loading, starting, and stopping of applications are sometimes supported. Mixed-time-criticality, where non-real-time and real-time applications coexist, is also often claimed. However, the levels of predictability, the formalisms used, and the level of automation vary considerably. Increasingly platforms claim to be composable, especially in the sense of temporal and space partitioning, but not to the extreme extent of CompSOC, i.e., no interference at the level of individual clock cycles.

In general, points on which platforms may be compared include the formalism for predictability, model(s) of computation offered, single or multiple applications, global or distributed arbitration, single-level or multi-level arbitration, work-conserving arbitration or not, use of budgets or virtual resources or not, time-triggered or data-driven execution, and distributed-shared-memory or message-passing architecture.

16.7 Conclusions

In this chapter, we first defined what a mixed-time-criticality system is and what its requirements are. After defining the concepts that such systems should follow, we described CompSOC, which is one example of a mixed-time-criticality platform. We described, in detail, how multiple resources, such as processors, memories, and interconnect, are combined into a larger hardware platform, and especially how they are shared between applications using different arbitration schemes. Following this, the software architecture that transforms the single hardware platform into multiple virtual execution platforms, one per application, was described.

Acknowledgments The development of CompSOC has been partially funded by European grants, including CATRENE CT217 RESIST; ARTEMIS 621429 EMC2, 621353 DEWI, 621439 ALMARVI, and ECSEL 692455 ENABLE-S3.

References

1. Akesson B, Goossens K (2011) Architectures and modeling of predictable memory controllers for improved system integration. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Grenoble. IEEE, pp 1–6
2. Akesson B, Goossens K (2011) Memory controllers for real-time embedded systems. Embedded systems series, 1st edn. Springer, New York
3. Akesson B, Hansson A, Goossens K (2009) Composable resource sharing based on latency-rate servers. In: Proceedings of Euromicro symposium on digital system design (DSD), Patras, pp 547–555
4. Akesson B, Molnos A, Hansson A, Ambrose Angelo J, Goossens K (2010) Composability and predictability for independent application development, verification, and execution. In: Hübner M, Becker J (eds) Multiprocessor system-on-chip – hardware design and tool integration, circuits and systems, chap. 2. Springer, Heidelberg, pp 25–56

5. Akesson B, Steffens L, Strooisma E, Goossens K (2008) Real-time scheduling using credit-controlled static-priority arbitration. In: Proceedings of international conference on embedded and real-time computing systems and applications (RTCSA). IEEE Computer Society, Washington, DC, pp 3–14
6. Beyranvand Nejad A, Molnos A, Goossens K (2013) A software-based technique enabling composable hierarchical preemptive scheduling for time-triggered applications. In: Proceedings of international conference on embedded and real-time computing systems and applications (RTCSA), Taipei
7. Bini E, Buttazzo G, Eker J, Schorr S, Guerra R, Fohler G, Arzen KE, Romero Segovia V, Scordino C (2011) Resource management on multicore systems: the ACTORS approach. Proc Microarch (MICRO) 31(1):72–81
8. Bolder J, Oomen T (2014) Rational basis functions in iterative learning control – with experimental verification on a motion system. IEEE Trans Control Syst Technol 23(2): 722–729
9. Chandrasekar K, Akesson B, Goossens K (2012) Run-time power-down strategies for real-time SDRAM memory controllers. In: Proceedings of design automation conference (DAC). ACM, New York, pp 988–993
10. Davis RI, Burns A (2011) A survey of hard real-time scheduling for multiprocessor systems. ACM Comput Surv (CSUR) 43(4):35
11. Edwards SA, Lee EA (2007) The case for the precision timed (pret) machine. In: Proceedings of the 44th annual design automation conference, New York. ACM, pp 264–265
12. Giannopoulou G, Stoimenov N, Huang P, Thiele L, de Dinechin BD (2015) Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. J Real-Time Syst 52(4):399–449
13. Goossens S, Akesson B, Goossens K (2013) Conservative open-page policy for mixed time-criticality memory controllers. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Grenoble, pp 525–530
14. Goossens K, Azevedo A, Chandrasekar K, Gomony MD, Goossens S, Koedam M, Li Y, Mirzoyan D, Molnos A, Beyranvand Nejad A, Nelson A, Sinha S (2013) Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. ACM Spec Interest Group Embed Syst (SIGBED) Rev 10(3):23–34
15. Goossens K, Hansson A (2010) The Aethereal network on chip after ten years: goals, evolution, lessons, and future. In: Proceedings of design automation conference (DAC), Anaheim, pp 306–311
16. Goossens K, Koedam M, Sinha S, Nelson A, Geilen M (2015) Run-time middleware to support real-time system scenarios. In: Proceedings of European conference on circuit theory and design (ECCTD), Trondheim
17. Goossens S, Kouters T, Akesson B, Goossens K (2012) Memory-map selection for firm real-time SDRAM controllers. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE). IEEE, Dresden, pp 828–831
18. Goossens S, Kuijsten J, Akesson B, Goossens K (2013) A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In: International conference on hardware/software codesign and system synthesis (CODES+ISSS), Montreal, pp 1–10
19. Hansson A, Ekerhult M, Molnos A, Milutinovic A, Nelson A, Ambrose J, Goossens K (2011) Design and implementation of an operating system for composable processor sharing. J Micromech Microeng (MICPRO) 35(2):246–260. Elsevier. Special issue on network-on-chip architectures and design methodologies
20. Hansson A, Goossens K (2007) Trade-offs in the configuration of a network on chip for multiple use-cases. In: Proceedings of international symposium on networks on chip (NOCS). IEEE Computer Society, Washington, DC, pp 233–242
21. Hansson A, Goossens K (2009) An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In: International conference on hardware/software codesign and system synthesis (CODES+ISSS). ACM, New York, pp 99–108

22. Hansson A, Goossens K (2010) On-Chip interconnect with aelite: composable and predictable systems. Embedded systems series. Springer, New York
23. Hansson A, Goossens K, Bekooij M, Huisken J (2009) CoMPSoC: a template for composable and predictable multi-processor system on chips. ACM Trans Des Autom Electron Syst 14(1):1–24
24. Hansson A, Wiggers M, Moonen A, Goossens K, Bekooij M (2009) Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. IET Comput Digit Tech 3(5):398–412
25. Henzinger TA, Horowitz B, Kirsch CM (2003) Giotto: a time-triggered language for embedded programming. Proc IEEE 91(1):84–99
26. Jansen B, Schwiegelshohn F, Koedam M, Duhem F, Masing L, Werner S, Huriaux C, Courtay A, Wheatley E, Goossens K, Lemonnier F, Millet P, Becker J, Sentieys O, Hübner M (2015) Designing applications for heterogeneous many-core architectures with the FlexTiles platform. In: Proceedings of International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), Samos
27. Kasapaki E, Sorensen RB, Müller C, Goossens K, Schoeberl M, Sparso J (2015) Argo: a real-time network-on-chip architecture with an efficient GALS implementation. IEEE Trans Very Large Scale Integr Syst (TVLSI) 99(2):479–492
28. Kirsch C, Sokolova A (2012) The logical execution time paradigm. In: Chakraborty S, Eberspächer J (eds) Advances in real-time systems (ARTS). Springer, Berlin/Heidelberg, pp 103–120
29. Kopetz H (2011) Real-time systems: design principles for distributed embedded applications. Springer, Heidelberg
30. Li Y, Salunkhe H, Bastos J, Moreira O, Akesson B, Goossens K (2015) Mode-controlled dataflow modeling of real-time memory controllers. In: Proceedings of embedded systems for real-time multimedia (ESTIMedia), Amsterdam
31. Moreira O, Corporaal H (2014) Scheduling real-time streaming applications onto an embedded multiprocessor. Embedded systems series, vol 24. Springer, Cham
32. Nejad AB, Molnos A, Goossens K (2013) A unified execution model for multiple computation models of streaming applications on a composable MPSoC. J Syst Archit (JSA) 59(10, part C), 1032–1046. Elsevier
33. Nejad AB, Molnos A, Martinez ME, Goossens K (2013) A hardware/software platform for QoS bridging over multi-chip NoC-based systems. J Parallel Comput (PARCO)39(9):424–441. Elsevier
34. Nelson A (2014) Composable and predictable power management. Ph.D. thesis, Delft University of Technology
35. Nelson A, Beyranvand Nejad A, Molnos A, Koedam M, Goossens K (2014) CoMik: a predictable and cycle-accurately composable real-time microkernel. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Dresden
36. Nelson A, Goossens K (2015) Distributed power management of real-time applications on a GALS multiprocessor SOC. In: Proceedings of ACM international conference on embedded software (EMSOFT), Amsterdam
37. Nelson A, Goossens K, Akesson B (2015) Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. J Syst Archit (JSA) 61(9):435–448
38. Nelson A, Molnos A, Goossens K (2011) Composable power management with energy and power budgets per application. In: Proceedings of international conference on embedded computer systems: architectures, modeling and simulation (SAMOS), Samos, pp 396–403
39. Nelson A, Molnos A, Goossens K (2012) Power versus quality trade-offs for adaptive real-time applications. In: Proceedings of embedded systems for real-time multimedia (ESTIMedia), Tampere, pp 75–84
40. Nesbit KJ, Smith JE, Moreto M, Cazorla FJ, Ramirez A, Valero M (2008) Multicore resource management. Proc Microarch (MICRO) 28(3):6–16

41. Nieuwland A, Kang J, Gangwal OP, Sethuraman R, Busá N, Goossens K, Peset Llopis R, Lippens P (2002) C-HEAP: a heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. ACM Trans Des Autom Embed Syst 7(3):233–270
42. Obermaisser R, Weber D (2014) Architectures for mixed-criticality systems based on networked multi-core chips. In: Proceedings of Conference on Emerging Technology and Factory Automation (ETFA), Barcelona, pp 1–10
43. Pelz G et al (2005) Automotive system design and autosar. In: Advances in design and specification languages for SoCs. Springer, New York, pp 293–305
44. Pinho LM, Nelis V, Yomsi PM, Quinones E, Bertogna M, Burgio P, Marongiu A, Scordino C, Gai P, Ramponi M, Mardiak M (2015) P-socrates: a parallel software framework for time-critical many-core systems. J Microprocess Microsyst 39(8):1190–1203. Elsevier
45. Rădulescu A, Dielissen J, González Pestana S, Gangwal OP, Rijpkema E, Wielage P, Goossens K (2005) An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. IEEE Trans CAD Integr Circuits Syst 24(1):4–17
46. Rushby J (1999) Partitioning in avionics architectures: requirements, mechanisms, and assurance. Technical report, NASA
47. Schoeberl M, Abbaspour S, Akesson B, Audsley N, Capasso R, Garside J, Goossens K, Goossens S, Hansen S, Heckmann R, Hepp S, Huber B, Jordan A, Kasapaki E, Knoop J, Li Y, Prokesch D, Puffitsch W, Puschner P, Rocha A, Silva C, Sparsø J, Tocchi A (2015) T-CREST: time-predictable multi-core architecture for embedded systems. J Syst Archit (JSA) 61(9):449–471. Elsevier
48. Sinha S, Koedam M, Breaban G, Nelson A, Nejad A, Geilen M, Goossens K (2015) Composable and predictable dynamic loading for time-critical partitioned systems on multiprocessor architectures. J Microprocess Microsyst (MICPRO) 39(8):1087–1107
49. Stefan R, Molnos A, Goossens K (2014) dAElite: a TDM NoC supporting QoS, multicast, and fast connection set-up. IEEE Trans Comput 63(3):583–594
50. Stuijk S, Basten T, Geilen M, Corporaal H (2007) Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In: Proceedings of design automation conference (DAC), San Diego, pp 777–782
51. Tavakoli R, Nabi M, Basten T, Goossens K (2015) Enhanced time-slotted channel hopping in wsns using non-intrusive channel-quality estimation. In: Proceedings of international conference on mobile ad hoc and sensor systems (MASS), Dallas
52. Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. In: The 2000 IEEE international symposium on circuits and systems, 2000. Proceedings. ISCAS 2000, Geneva, vol 4. IEEE, pp 101–104
53. Trujillo S, Crespo A, Alonso A, Perez J (2014) Multipartes: multi-core partitioning and virtualization for easing the certification of mixed-criticality systems. J Microprocess Microsyst 38(8, part B):921–932. Elsevier
54. Ungerer T, Bradatsch C, Gerdes M, Kluge F, Jahr R, Mische J, Fernandes J, Zaykov PG, Petrov Z, Boddeker B, Kehr S, Regler H, Hugl A, Rochange C, Ozaktas H, Casse H, Bonenfant A, Sainrat P, Broster I, Lay N, George D, Quinones E, Panic M, Abella J, Cazorla F, Uhrig S, Rohde M, Pyka A (2013) parmerasa – multi-core execution of parallelised hard real-time applications supporting analysability. In: Proceedings of Euromicro symposium on digital system design (DSD), Los Alamitos
55. Valencia J, van Horsen E, Goswami D, Heemels M, Goossens K (2016) Resource utilization and quality-of-control trade-off for a composable platform. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Lausanne
56. Windsor J et al (2009) Time and space partitioning in spacecraft avionics. In: SMC-IT, Pasadena
57. Zhang H (1995) Service disciplines for guaranteed performance service in packet-switching networks. Proc IEEE 83(10):1374–1396

Part V
Hardware/Software Cosimulation and Prototyping

Parallel Simulation

17

Rainer Dömer, Guantao Liu, and Tim Schmidt

Abstract

The SystemC standard is widely used in industry and academia to model and simulate electronic system-level designs. However, despite the availability of multi-core processor hosts, the reference SystemC simulator is still based on sequential *Discrete Event Simulation (DES)* which executes only a single thread at any time.

In recent years, parallel SystemC simulators have been proposed which run multiple threads in parallel based on *Parallel Discrete Event Simulation (PDES)* semantics. While this can improve the simulator run time by an order of magnitude, synchronous PDES requires careful dependency analysis of the model and still limits the parallel execution to threads that run at the same simulation time.

In this chapter, we review the classic DES and PDES algorithms and then present a state-of-the-art approach called *Out-of-Order Parallel Discrete Event Simulation (OOO PDES)* which breaks the traditional time cycle barrier and executes threads in parallel and out of order (ahead of time) while maintaining the standard SystemC modeling semantics. Specifically, we present our *Recoding Infrastructure for SystemC (RISC)* that consists of a dedicated SystemC compiler and advanced parallel simulator. RISC provides an open-source reference implementation of OOO PDES and achieves fastest simulation speed for traditional SystemC models without any loss of accuracy.

R. Dömer (✉)
Center for Embedded and Cyber-physical Systems, Department of Electrical Engineering and Computer Science, The Henry Samueli School of Engineering, University of California, Irvine, CA, USA
e-mail: doemer@uci.edu

G. Liu • T. Schmidt
Center for Embedded and Cyber-physical Systems, University of California, Irvine, CA, USA
e-mail: guantaol@uci.edu; schmidtt@uci.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_19

Acronyms	
AST	Abstract Syntax Tree
DE	Discrete Event
DES	Discrete Event Simulation
DUT	Design Under Test
ESL	Electronic System Level
OOO PDES	Out-of-Order Parallel Discrete Event Simulation
PDES	Parallel Discrete Event Simulation
RISC	Recoding Infrastructure for SystemC
SG	Segment Graph
SLDL	System-Level Description Language

Contents

17.1 Introduction

Electronic System Level (ESL) design space exploration in general and the validation of ESL designs in particular typically rely on dynamic *model simulation* in order to study and optimize a desired system or electronic device before it is actually being built. For this simulation, an abstract model of the intended system is first specified in a System-Level Description Language (SLDL) such as SpecC [13] or SystemC [15]. This SLDL model is then compiled, executed, and evaluated

on a host computer. In order to reflect causal ordering and provide abstract timing information, SLDL simulation algorithms are usually based on the classic *Discrete Event (DE)* semantics which drive the simulation process forward by use of events and simulation time advances.

As an IEEE standard [18], the SystemC SLDL [15] is predominantly used in both industry and academia. Under the umbrella of the Accellera Systems Initiative [1], the SystemC Language Working Group [31] maintains not only the official SystemC language definition but also provides an open-source proof-of-concept library [16] that can be used free of charge to simulate SystemC design models within a standard C++ programming environment. However, this reference implementation follows the traditional scheme of sequential *Discrete Event Simulation (DES)* which is much easier to implement than truly parallel approaches. Here, the discrete time events generated during the simulation are processed strictly in sequential order. As such, the SystemC reference simulator runs fully sequentially and cannot utilize any parallel computing resources available on today's multi- and many-core processor hosts. This severely limits the simulation speed since a single processor core has to perform all the work.

17.1.1 Exploiting Parallelism for Higher Simulation Speed

In order to provide faster simulation performance, parallel execution is highly desirable, especially since the SLDL model itself already contains clearly exposed parallelism that is explicitly specified by the language constructs, such as `SC_METHOD`, `SC_THREAD`, and `SC_CTHREAD` in the case of SystemC SLDL. Here, the topic of *Parallel Discrete Event Simulation (PDES)* [12] has recently gained a lot of attraction again. Generally, the PDES approach issues multiple threads concurrently and runs these threads in parallel on the multiple available processor cores. In turn, the simulation speed increases significantly. The naive assumption of a linear speedup of factor $n x$ for n available processor cores occurs very rarely due to the usually quite limited amount of parallelism exposed in the model and the application of Amdahl's law [2]. Typically, a speedup of one order of magnitude can be expected for embedded system applications with parallel modules [6].

With respect to PDES performance, it is very important to understand the dependencies among the discrete time events and also to distinguish between the dependencies present in the model and the (additional) dependencies imposed by the simulation algorithm. In general, SLDL models assume a *partial order* of events where certain events depend on others due to a causal relationship or occurrence in sequence along the time line, and other events are independent and may be performed in any order or in parallel. On top of this required partial ordering of events in the model, a sequential simulator (described in Sect. 17.2) imposes a much stronger *total order* on the event processing. This is acceptable as it observes the weaker SLDL execution semantics but results in slow performance due to the overly restrictive sequential execution.

In contrast, PDES follows a partial ordering of events. Here, we can distinguish between regular *synchronous PDES*, which advances the (global) simulation time in

order and executes threads in lockstep parallel fashion, and aggressive *asynchronous PDES*, where simulation time is effectively localized and independent threads can execute out of order and run ahead of others. As we will explain in detail in Sect. 17.3, the strict in-order execution imposed by synchronous PDES still limits the opportunities for parallel execution. When a thread finishes its evaluation phase, it has to wait until all other threads have completed their evaluation phases as well. Threads finishing early must stop at the simulation cycle barrier, and available processor cores are left idle until all runnable threads reach the cycle barrier.

This problem is overcome by the latest state-of-the-art approaches which implement a novel technique called *Out-of-Order Parallel Discrete Event Simulation (OOO PDES)* [6, 8]. By internally localizing the simulation time to individual threads and carefully processing dependent events only at required times, an OOO PDES simulator can issue threads in parallel and ahead of time, exploiting the maximum available parallelism without loss of accuracy. Thus, the OOO PDES presented in Sect. 17.4 minimizes the idle time of the available parallel processor cores and results in highest simulation speed while fully maintaining the standard SLDL and unmodified model semantics.

We should note that parallel simulation in general, and synchronous PDES and OOO PDES in particular, can be implemented in different environments and are *not* language dependent. In fact, the parallel simulation techniques discussed in Sects. 17.3 and 17.4 have been originally designed using the SpecC language [10, 13, 35] and integrated into the system-on-chip environment (see ▸ Chap. 31, "SCE: System-on-Chip Environment" and in particular Sect. 4.1, "Simulation") but can be equally well applied to other SLDLs with explicitly exposed parallelism. Without loss of generality, and in order to present the state of the art, we describe in this chapter the evolution from sequential DES over synchronous PDES to asynchronous OOO PDES using the SystemC SLDL [15, 18] which is both the de facto and official standard for ESL design today. In particular, we describe in Sect. 17.5 our Recoding Infrastructure for SystemC (RISC) [21], which is open source [20] and consists of a dedicated SystemC compiler and corresponding out-of-order parallel simulator, and thus provides a reference implementation of OOO PDES for SystemC.

Finally, we conclude this chapter in Sect. 17.6 with experimental results for embedded application examples that demonstrate the significantly reduced simulator run times due to parallel simulation.

17.1.2 Related Work on Accelerated Simulation

Parallel simulation is a well-studied subject in the literature [4, 12, 24]. Two major synchronization paradigms can be distinguished, namely, conservative and optimistic [12]. *Conservative* PDES typically involves dependency analysis and ensures in-order execution for dependent threads. In contrast, the *optimistic* paradigm assumes that threads are safe to execute and rolls back when this proves incorrect. Often, the temporal barriers in the model prevent effective parallelism in conservative PDES while rollbacks in optimistic PDES are expensive in execution.

The OOO PDES presented in Sect. 17.4 is conservative and can be seen as an improvement over synchronous PDES (Sect. 17.3) approaches, such as [11, 27] for SystemC and [7] for SpecC.

Distributed parallel simulation, including [4, 17], is a natural extension of PDES where the system model is partitioned into sets of modules and distributed onto a set of networked host computers. Here, the necessary model partitioning may create extra work, and often the network speed becomes a bottleneck for synchronization and communication among the multiple host simulators.

In a wider perspective, simulation performance can also be improved by clever software modeling and utilizing specialized hardware. Software approaches include the overall concept of transaction-level modeling (TLM) [3] and temporal decoupling [32, 33], which abstract communication from a pin-accurate level to entire transactions with only approximate timing, and source-level [30] or host-compiled simulation [14], which model computation and scheduling at a higher abstraction level. Typically, these modeling techniques trade off higher simulation speed at the cost of reduced timing accuracy.

Simulation acceleration can also be achieved by special purpose hardware, such as field-programmable gate arrays (FPGA) and graphics processing units (GPU). For example, [29] emulates SystemC code on FPGA boards and [23] proposes a SystemC multi-threading model on GPUs. As a hybrid approach, [28] parallelizes SystemC simulation across multiple CPUs and GPUs. Usually, special hardware approaches pose limitations on the model partitioning across the heterogeneous simulator units.

Parallel simulation can also be organized as multiple simulators which run independently in parallel and synchronize regularly or when needed (i.e., when timing differences grow beyond a given threshold). For instance, the Wisconsin Wind Tunnel [22] uses a conservative time bucket synchronization scheme to coordinate multiple simulators at predefined intervals. Another example [34] introduces a simulation backplane to handle the synchronization between wrapped simulators and analyzes the system to optimize the period of synchronization message transfers.

From this discussion, we can see that parallel simulation is an important topic that has been addressed and optimized with a multitude of approaches. For the remainder of this book chapter, we will focus on the SystemC language and present two general parallel simulation techniques, namely, synchronous PDES in Sect. 17.3 and an advanced out-of-order PDES algorithm in Sect. 17.4. Both approaches are general in the sense that they do not require any special setup or hardware, pose no limitations on the simulation model, and do not sacrifice any timing accuracy in the result.

17.2 Discrete Event Simulation (DES)

Before we address the details of modern parallel simulation, we review in this section the classic sequential simulation of models with discrete time. After the description of this traditional DES algorithm, we then extend it incrementally to synchronous PDES in Sect. 17.3 and finally to out-of-order PDES in Sect. 17.4.

17.2.1 Discrete Time and Discrete Event Model

SLDL simulation is driven by discrete events and simulation time advances. We will use the term *simulation time* for the simulated time maintained by the simulator. This must not be confused with *simulator run time* which is the actual wall-clock time that measures how long it takes the simulator to perform the simulation on the host. Simulation time consists of a tuple (t, δ) where t represents an integral amount of simulated time since the simulation start $t = 0$ and δ is a positive integer that counts iterations at the same simulated time t due to event notifications.

Formally, time tuples, often referred to as time stamps, form a partial order and can be compared as follows [6]:

equal: $(t_1, \delta_1) = (t_2, \delta_2)$ iff $t_1 = t_2, \delta_1 = \delta_2$
before: $(t_1, \delta_1) < (t_2, \delta_2)$ iff $t_1 < t_2$, or $t_1 = t_2, \delta_1 < \delta_2$
after: $(t_1, \delta_1) > (t_2, \delta_2)$ iff $t_1 > t_2$, or $t_1 = t_2, \delta_1 > \delta_2$

Events serve the purpose of synchronization among communicating or dependent threads. Threads can *notify* events or *wait* for events. An event notification at time (t, δ) reaches a thread waiting for the event at the same time, or it expires without any effect if no thread is waiting for the event. If a thread wakes up due to a notified event, it resumes its execution at the next delta increment $(t, \delta + 1)$ or immediately at (t, δ) in case of a SystemC immediate notification.

In other words, the SLDL semantics use an outer cycle in the simulation process to model time advances so that thread execution can reflect estimated duration or delays by adding to time t. In addition, there is an inner cycle in the simulator, called *delta cycle*, that is used for event notifications which may in turn wake waiting threads, in which case time is incremented by one δ count. In the case of SystemC, yet another innermost cycle is available where so-called immediate event notifications may take place without any time advance. Since immediate notifications can easily lead to nondeterministic models with potential deadlocks or lost events, these should generally be avoided.

17.2.2 Scheduling Queues

SLDLs use a set of parallel threads to execute the functionality in the model. In SystemC, such threads are explicitly specified as `SC_METHOD`, `SC_THREAD`, or `SC_CTHREAD`. These threads are then managed by a scheduler in the simulator kernel which decides when threads are actually dispatched to run.

To coordinate the execution of the threads according to the SLDL semantics, the simulation scheduler typically maintains sorted lists or *queues* of threads where each thread is a member of one queue at any time. For ease of understanding, we use a simplified formal model here and ignore special cases in the SystemC language,

such as suspended processes. Formally we define the following scheduling sets [9]:

$\mathsf{THREADS} = \mathsf{READY} \cup \mathsf{RUN} \cup \mathsf{WAIT} \cup \mathsf{WAITTIME}$
$\mathsf{READY} = \{\ th \mid th \text{ is ready to run}\ \}$
$\mathsf{RUN} = \{\ th \mid th \text{ is currently running}\ \}$
$\mathsf{WAIT} = \{\ th \mid th \text{ is waiting for one or more events}\ \}$
$\mathsf{WAITTIME} = \{\ th \mid th \text{ is waiting for time advance}\ \}$

At the beginning of simulation at time $(0, 0)$, all threads are placed into the READY queue and the other sets are empty:

$\mathsf{THREADS} = \mathsf{READY}$
$\mathsf{RUN} = \mathsf{WAIT} = \mathsf{WAITTIME} = \varnothing$

During simulation, the scheduler moves threads between the queues and suspends or resumes their execution according to their state. In order to describe the discrete event scheduling algorithms below, we formally define the following scheduling operations on threads *th* maintained in queues A and B:

Run(*th*): thread *th* begins or resumes its execution
Stop(): the current thread stops running or suspends its execution
Yield(*th*): the current thread stops running and yields execution to thread *th*
th = **Pick**(A): pick one thread *th* out of set A
Move(*th*, A, B): move thread *th* from set A to set B

For inter-thread synchronization and communication, the SystemC language provides events and channel primitives with special **update** semantics. Without going into the details of event notifications and channel updates, we denote the set of instantiated primitive channels in the model as CHANNEL.

17.2.3 Sequential Discrete Event Scheduler

Figure 17.1 shows the traditional DES scheduling algorithm as it is implemented by the Accellera SystemC reference simulator [16]. Most notably, this algorithm is fully sequential in the sense that only a single thread is made runnable at all times. In SystemC specifically, the choice of the next thread to run is nondeterministic by definition, so one thread is randomly picked from the READY queue and placed into the RUN queue for execution. When the thread returns to the scheduler due to execution of a `wait` statement, it yields control back to the scheduler which in turn picks the next thread to run.

When the READY queue is empty, the scheduler performs requested channel updates and event notifications which typically fills the READY queue again with threads that wake up due to events they were waiting for. These are taken out of the WAIT queue and a new delta cycle begins.

If no threads become ready after the update and notification phase, the current time cycle is complete. Then the scheduler advances the simulation time, moves all

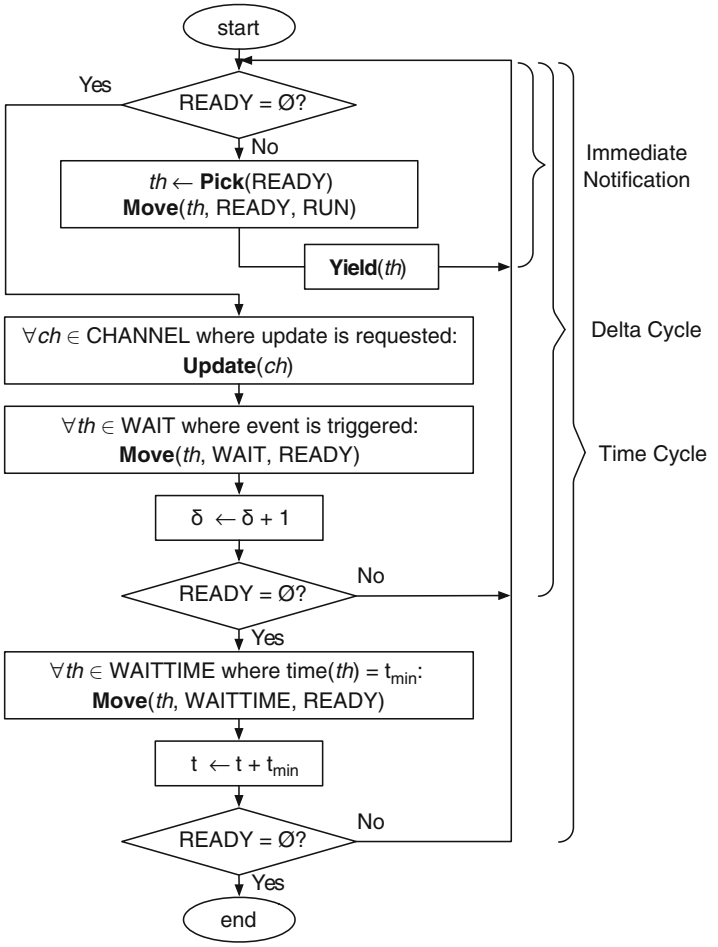


Fig. 17.1 Traditional discrete event simulation (DES) scheduler for SystemC

threads with the earliest next time stamp from the WAITTIME queue into the READY queue, and resumes execution with the next time cycle.

Finally, when both the READY and WAITTIME queues are empty, the simulation terminates.

17.3 Parallel Discrete Event Simulation (PDES)

The sequential DES algorithm can be easily extended to support synchronous parallel simulation. Instead of a single thread in DES, regular PDES manages multiple threads at the same time in the RUN queue. These threads can then execute truly in parallel on the parallel processors of the host.

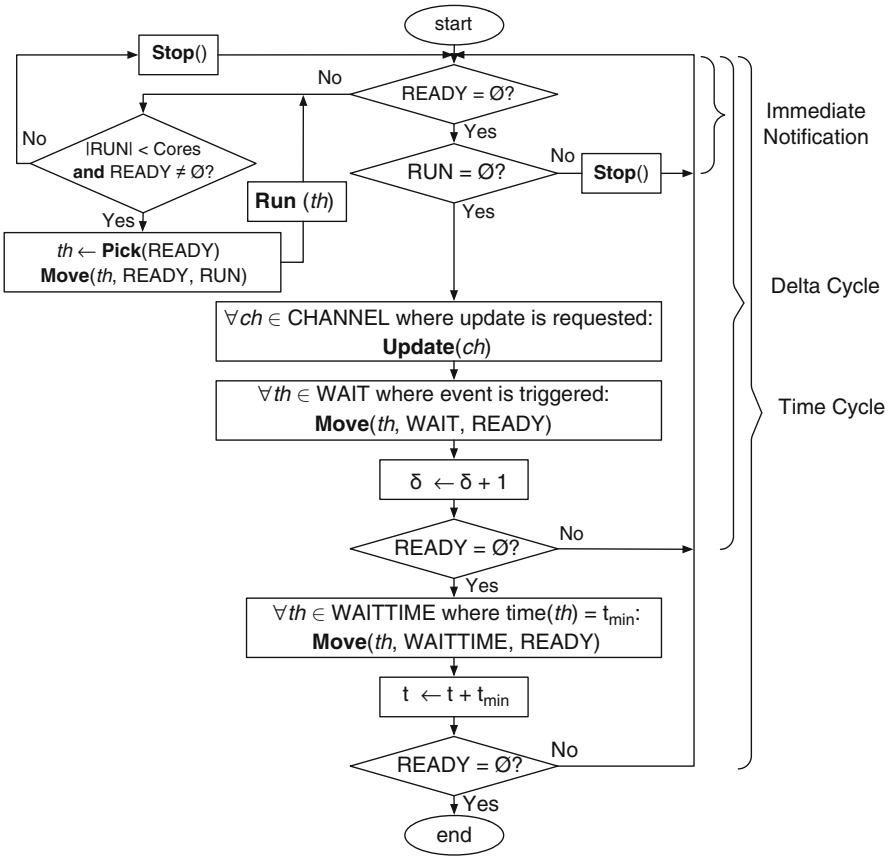


Fig. 17.2 Synchronous parallel discrete event simulation (PDES) scheduler for SystemC

17.3.1 Parallel Discrete Event Scheduler

The PDES scheduling algorithm, as shown in Fig. 17.2, operates the same way as the traditional scheduler in Fig. 17.1, but with one exception: the synchronous parallel scheduler picks multiple threads from the READY queue and runs them in parallel on the available processor cores.

In the evaluation phase, as long as the READY queue is not empty and an idle core is available, the PDES scheduler issues a new thread from the READY queue in a loop. When a thread finishes earlier than the other threads in the same cycle, a new ready thread is picked and assigned to the free processor core. Thus, within the same delta cycle, PDES keeps as many processor cores as busy as possible.

However, we notice that only threads at the same time (t, δ) run in parallel. Synchronous PDES implies an absolute barrier at the end of each delta and time cycle. All threads need to wait at the barrier until all other runnable threads finish

their current evaluation phase. Only then the synchronous scheduler performs the channel update and event notification phase for the next delta or advances simulation time for the next time cycle.

17.3.2 Protection of the Parallel Simulation Kernel

The benefit of PDES running more than a single thread at the same time comes at a price. Explicit synchronization among the parallel threads becomes necessary in critical sections. In particular, shared data structures in the simulation kernel, including the global simulation time, event lists, and thread queues, need to be properly protected for mutual exclusive access by the concurrent threads.

In order to protect the central scheduling resources, locks (binary semaphores) and condition variables need to be introduced for proper thread synchronization. For example, our RISC prototype implementation (see Sect. 17.5 below) uses one dedicated kernel lock to protect the scheduling resources. This lock is acquired by the threads at every kernel entry and released upon kernel exit. Each thread also owns a condition variable c that is used in combination with the lock to put threads to sleep (**Stop**() calls `wait`(c)) or wake them up (**Run**(th) calls `signal`(c)), as determined by the scheduling kernel.

17.3.3 Preserving SystemC Execution Semantics in PDES

In contrast to the SpecC language, which allows preemptive parallel execution of the threads in the model [10], the SystemC language poses strict rules on standard-compliant simulation. This is a very important aspect to consider when applying PDES to SystemC. For semantics-compliant SystemC simulation, complex inter-dependency analysis over all threads and variables in the model is a prerequisite to parallel execution [9]. The IEEE standard SystemC Language Reference Manual (LRM) [18] clearly states that "process instances execute without interruption" and presumably is meant to simplify the writing of SystemC models. Here, the need to prevent parallel access conflicts to shared variables and to avoid potential race conditions among the parallel threads becomes a burden for the simulation environment (rather than for the model designer).

This requirement is also known as *cooperative (or coroutine) multitasking* which is explicitly assumed by the SystemC execution semantics. As detailed in [9], the particular problem of parallel simulation is also explicitly addressed in the SystemC LRM [18]:

> An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the coroutine semantics defined in this subclause. In other words, the implementation would be obliged to analyze any dependencies between processes and constrain their execution to match the coroutine semantics.

Consequently, a standard-compliant PDES environment for SystemC must identify and resolve any dependencies among the threads in the model. We will describe this required dependency analysis in detail in Sect. 17.5.3 because it is needed for both synchronous and out-of-order PDES.

17.4 Out-of-Order Parallel Discrete Event Simulation (OOO PDES)

In OOO PDES [6], we break the synchronous barrier in the simulator so that independent threads can also run in parallel when they are at a different simulation times (different t or different δ). In other words, threads are allowed to run ahead in time and thus can execute out-of-order, unless a causal relationship prohibits it.

17.4.1 Thread-Local Simulation Time

For OOO PDES, we replace the global simulation time (t, δ) with local time stamps for each thread. Thus, each thread th maintains its own time (t_{th}, δ_{th}).

Events get assigned their own time, too. Since events in the simulation model can occur multiple times and at different simulation times, we note an event e notified at time (t, δ) as a triple (id_e, t_e, δ_e). Thus, every event is managed with its own ID and notification time attached.

Finally, we distinguish the sets of events that have been notified at a given time. Formally, we define:

$$\mathsf{EVENTS} = \cup \mathsf{EVENTS}_{t,\delta}$$
$$\mathsf{EVENTS}_{t,\delta} = \{(id_e, t_e, \delta_e) \mid t_e = t, \delta_e = \delta)\}$$

17.4.2 Dynamically Evolving Scheduling Queues

Rather than the static DES queues which exist at all times, we define for OOO PDES multiple sets ordered by their local time stamps and dynamically create and delete these sets as needed. For efficiency reasons, these sets are typically implemented as true queues where the threads are ordered by increasing time stamps. Formally, we define the following queues:

$\mathsf{QUEUES} = \{\mathsf{READY}, \mathsf{RUN}, \mathsf{WAIT}, \mathsf{WAITTIME}\}$
$\mathsf{READY} = \cup \mathsf{READY}_{t,\delta}$, where $\mathsf{READY}_{t,\delta} = \{th \mid th \text{ is ready to run at } (t, \delta)\}$
$\mathsf{RUN} = \cup \mathsf{RUN}_{t,\delta}$, where $\mathsf{RUN}_{t,\delta} = \{th \mid th \text{ is running at } (t, \delta)\}$
$\mathsf{WAIT} = \cup \mathsf{WAIT}_{t,\delta}$, where $\mathsf{WAIT}_{t,\delta} = \{th \mid th \text{ is waiting since } (t, \delta) \text{ for events } (id_e, t_e, \delta_e), (t_e, \delta_e) \geq (t, \delta)\}$
$\mathsf{WAITTIME} = \cup \mathsf{WAITTIME}_{t,\delta}$, where $\delta = 0$, $\mathsf{WAITTIME}_{t,\delta} = \{th \mid th \text{ is waiting for simulation time advance to } (t, 0)\}$

As in the regular DES case, the simulation starts at time $(0, 0)$ with all threads in the $\mathsf{READY}_{0,0}$ queue. Then again the threads change state by transitioning between the queues, as determined by the scheduler:

Move$(th, \mathsf{READY}_{t,\delta}, \mathsf{RUN}_{t,\delta})$: thread th is issued and becomes runnable
Move$(th, \mathsf{RUN}_{t,\delta}, \mathsf{WAIT}_{t,\delta})$: thread th calls `wait`(e) for an event e
Move$(th, \mathsf{RUN}_{t,\delta}, \mathsf{WAITTIME}_{t',0})$, where $t < t' = t + d$: thread th calls `wait`(d) to wait for a time delay d
Move$(th, \mathsf{WAIT}_{t,\delta}, \mathsf{READY}_{t',\delta''})$, where $(t, \delta) \leq (t', \delta'')$: thread th is waiting since time (t, δ) for event $e = (id_e, t'_e, \delta'_e)$ which is notified at time (t', δ'); in turn, thread th becomes ready to run at (t', δ'') where $\delta'' = \delta'$ (immediate notification) or $\delta'' = \delta' + 1$ (regular delta cycle notification)
Move$(th, \mathsf{WAITTIME}_{t,\delta}, \mathsf{READY}_{t,\delta})$, where $\delta = 0$: simulation time advances to time $(t, 0)$, making one or more threads th ready to run; the local time for these threads th is set to (t_{th}, δ_{th}) where $t_{th} = t$ and $\delta_{th} = 0$

Whenever the sets $\mathsf{READY}_{t,\delta}$ and $\mathsf{RUN}_{t,\delta}$ become empty and there are no $\mathsf{WAIT}_{t',\delta'}$ or $\mathsf{WAITTIME}_{t',\delta'}$ queues with earlier time stamps $(t', \delta') \leq (t, \delta)$, then the scheduler can delete these sets as well as any expired events $\mathsf{EVENTS}_{t,\delta}$.

17.4.3 Out-of-Order Parallel Discrete Event Scheduler

Figure 17.3 shows the OOO PDES scheduling algorithm. Since each thread maintains its own local time, the scheduler can relax the nested loops structure of synchronous PDES and deliver events and update simulation times individually, providing more flexibility for threads to run in parallel. Overall, this results in a higher degree of parallelism and thus higher simulation speed.

Note that the prior loops for explicit delta cycles and time cycles in the scheduler control flow do not exist any more for OOO PDES. Instead, we only have one main loop where both the notification phase and time updates are processed. The READY queue is consequently filled with more threads which, however, are now subject to possible conflicts. These conflicts are then taken into account when threads are picked from the READY queue and issued for execution into the RUN set.

Note also that the WAITTIME queue gets cleared in every scheduling step and all the threads move into the timed READY queue. Then, when the scheduler picks ready threads to run, it prefers earlier ones over threads with later time stamps. This order prevents threads from starving in the READY queue and also minimizes conflicts among the ready threads.

Potential conflicts are strictly averted by the NoConflicts(th) condition in Fig. 17.3 when runnable threads are picked. Here, detailed dependency analysis is used to avoid potential data, event, and time advance hazards among the set of threads in RUN that are executing in parallel. Only if a thread th has NoConflicts(th) it can be issued for parallel execution.

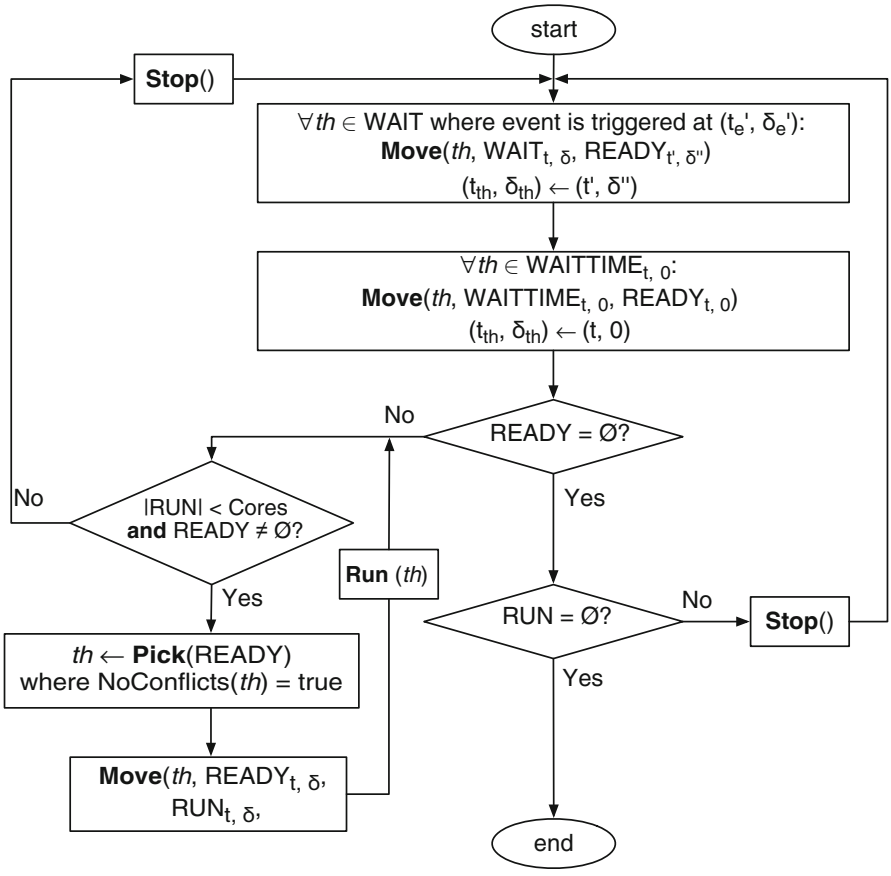


Fig. 17.3 Out-of-order parallel discrete event simulation (OOO PDES) scheduler for SystemC

17.4.4 OOO PDES Scheduling Algorithm

Algorithm 3 formally defines the scheduling algorithm of OOO PDES. At each scheduling step, the scheduler first evaluates notified events and wakes up corresponding threads from WAIT. If a thread receives its event e with time stamp (t'_e, δ'_e), it becomes ready to run and its local time advances to either (t'_e, δ'_e) for an immediate notification or $(t'_e, \delta'_e + 1)$ for regular delta notifications.

After event notifications, the scheduler processes local time advances and moves any threads in WAITTIME to the $\mathsf{READY}_{t,0}$ queue corresponding to their wait time.

Then the scheduler issues threads for parallel execution as long as idle CPU cores and threads without conflicts are available. Finally, if no threads can run, that is when $\mathsf{RUN} = \mathsf{READY} = \emptyset$, the simulator terminates.

Note that Algorithm 3 allows to enable/disable the parallel out-of-order execution at any time by setting the Cores parameter. For example, when in-order execution is

Algorithm 3 OOO PDES scheduling algorithm

```
1: procedure OOOPDES_SCHEDULER
2:     for all th ∈ WAIT do                                   ▷ Process event notifications
3:         if ∃e = (id_e, t'_e, δ'_e) where th awaits e and (t'_e, δ'_e) ≥ (t, δ) then
4:             if e is an immediate notification then
5:                 Move(th, WAIT_{t,δ}, READY_{t'_e,δ'_e})
6:                 t_th ← t'_e; δ_th ← δ'_e
7:             else
8:                 Move(th, WAIT_{t,δ}, READY_{t'_e,δ'_e+1})
9:                 t_th ← t'_e; δ_th ← δ'_e + 1
10:            end if
11:        end if
12:    end for
13:    for all th ∈ WAITTIME do                               ▷ Process local time advances
14:        Move(th, WAITTIME_{t,δ}, READY_{t,δ})
15:        t_th ← t; δ_th ← 0
16:    end for
17:    for all th ∈ READY do                                  ▷ Out-of-order evaluation phase
18:        if |RUN| < Cores and NOCONFLICTS(th) then
19:            Run(th)
20:        end if
21:    end for
22:    return                                                 ▷ End of simulation
23: end procedure
```

needed for debugging purposes, we can set Cores to 1, and the algorithm will behave the same way as the traditional DES where only one thread is running in order at all times.

OOO PDES relies heavily on efficient conflict detection. At run time, the scheduler calls the function NoConflicts(*th*) listed in Algorithm 4. NoConflicts(*th*) checks for potential conflicts with all concurrent threads in the RUN and READY queues that run at an earlier time than the candidate thread *th*. For each concurrent thread, function Conflict(th_1, th_2) checks for any data, time, and event hazards. We will explain these hazards and their analysis in detail in Sect. 17.5.3 below, because we can rely on the compiler to carry the heavy burden of this complex analysis and pass prepared conflict tables to the simulator. At run time, the scheduler can then perform these checks in constant time (O(1)) by use of table lookups.

17.5 Recoding Infrastructure for SystemC (RISC)

We have realized the OOO PDES approach for the SystemC language with our RISC. This proof-of-concept prototype environment consists of a compiler and simulator with examples and documentation. The RISC software package is available as open source on our website [20] and can be installed on any regular multi-core Linux host.

Algorithm 4 Conflict detection in OOO PDES scheduler

```
1: function NoConflicts(th)
2:     for all th_2 ∈ RUN_{t,δ} ∪ READY_{t,δ} where (t, δ) < (t_th, δ_th) do
3:         if Conflict(th, th_2) then
4:             return false
5:         end if
6:     end for
7:     return true
8: end function

9: function Conflict(th_1, th_2)
10:    if th_2 has data conflicts with th_1 then                ▷ check data hazards
11:        return true
12:    end if
13:    if th_2 may enter another segment before th_1 then       ▷ check time hazards
14:        return true
15:    end if
16:    if th_2 may wake up another thread th_3 to run before th_1 then   ▷ check event hazards
17:        return true
18:    end if
19:    return false
20: end function
```

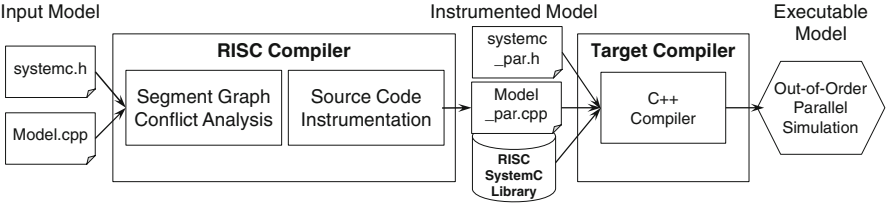


Fig. 17.4 RISC compiler and simulator for out-of-order PDES of SystemC

To perform semantics-compliant parallel SystemC simulation with out-of-order scheduling, we introduce a dedicated SystemC compiler that works hand in hand with a new simulator. This is in contrast to the traditional SystemC simulation flow where a SystemC-agnostic C++ compiler includes the SystemC headers and links the input model directly against the reference SystemC library.

As shown in Fig. 17.4, our RISC compiler acts as a frontend that processes the input SystemC model and generates an intermediate model with special instrumentation for OOO PDES. The instrumented parallel model is then linked against the extended RISC SystemC library by the target compiler (a regular C++ compiler) to produce the final executable output model. OOO PDES is then performed simply by running the generated executable model.

From the user perspective, we simply replace the regular C++ compiler with the SystemC-aware RISC compiler (which in turn calls the underlying C++ compiler). Otherwise, the overall SystemC validation flow remains the same as before. It will be just faster due to the parallel simulation.

Internally, the RISC compiler performs three major tasks, namely, segment graph construction, conflict analysis, and source code instrumentation.

17.5.1 Segment Graph

The first task of the RISC compiler is to parse the SystemC input model into an Abstract Syntax Tree (AST). Since SystemC is syntactically regular C++ code, RISC relies here on the ROSE compiler infrastructure [25]. The ROSE internal representation (IR) provides RISC with a powerful C/C++ compiler foundation that supports AST generation, traversal, analysis, and transformation.

As illustrated with the RISC software stack shown in Fig. 17.5a, the RISC compiler then builds on top of the ROSE IR a SystemC internal representation which accurately reflects the SystemC structures, including the module and channel hierarchy, port connectivity, and other SystemC-specific constructs. Up until this layer, the RISC software stack is very similar to the SystemC-clang framework [19].

On top of this, the RISC compiler then builds a segment graph data structure. A *Segment Graph (SG)* [8] is a directed graph that represents the code segments executed by the threads in the model. With respect to SystemC simulation, these segments are separated by the scheduler entry points, i.e., the `wait` statements in the SystemC code. In other words, the discrete events in the SystemC execution semantics are explicitly reflected in the SG as segment boundaries.

Note that a general segment graph may be defined with different segment boundaries. In fact, the RISC infrastructure takes the segment boundary as a flexible parameter that may be set to any construct found in the code, including function calls or control flow statements, such as `if`, `while`, or `return`. Here, we use `wait` statements as boundary (specified as "segment graph [`wait`]" in Fig. 17.5c)

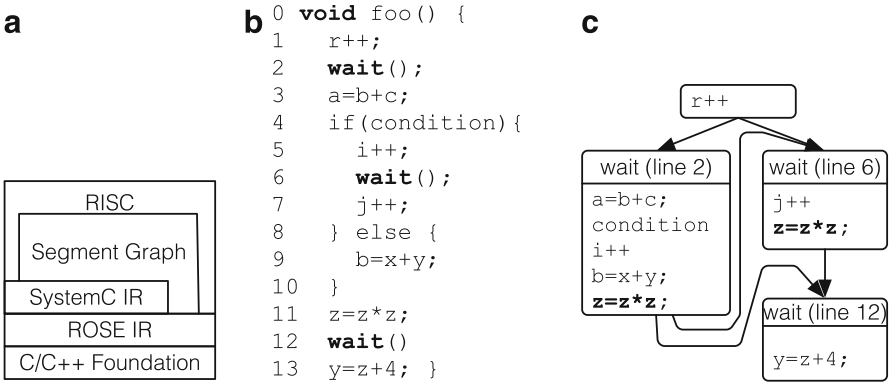


Fig. 17.5 Recoding infrastructure for SystemC (RISC) and a segment graph (SG) example [26]. (**a**) RISC software stack. (**b**) Example source code. (**c**) Segment graph [wait]

since for the purpose of parallel simulation we are interested in the segments of code executed between two scheduling points.

Formally, a segment graph consists of nodes and edges [6]. While the nodes are defined by the specified boundaries, the edges in the SG are defined by the possible control flow transitions. A transition exists between two segment nodes S_1 and S_2 if the flow of control starting from segment S_1 can reach segment S_2.

For example, the SystemC source code in Fig. 17.5b results in the SG shown in Fig. 17.5c where the segment boundary is chosen as SystemC `wait`. Here, the control flow from the start can reach the two `wait` statements in lines 2 and 6, resulting in the two edges to segment "wait(line2)" and "wait(line6)." Note also that source code lines may become part of multiple segments. Here, the assignment `z=z*z` is part of both segments "wait(line2)" and "wait(line6)" because it can be reached from both nodes.

17.5.2 Segment Graph Construction

The automatic construction of a segment graph is a complex process for the compiler. In this section, we first outline the main aspects and then provide a formal algorithm for the SG generation.

In contrast to the initial SpecC-based implementation [6, 8] (which is part of ▸Chap. 31, "SCE: System-on-Chip Environment") which has had several limitations, the RISC SG generator can build a graph from any given scope in a C/C++-based code and the user can freely choose the segment boundaries (as stated above for a general SG). There are also no control flow limitations. The RISC compiler fully supports recursive functions, jump statements `break` and `continue`, as well as multiple `return` statements from functions. Finally, expressions with an undefined evaluation order will be properly ordered to avoid ambiguity.

Algorithm 5 formally defines the central function BUILDSG used in the RISC SG generator. Function BUILDSG is recursive and builds the graph of segments by traversing the AST of the design model. Here, the first parameter *CurrStmt* is the current statement which is processed next. The set *CurrSegs* contains the current set of segments that lead to *CurrStmt* and thus will incorporate the current statement. For instance, while processing the assignment `z=z*z` in Fig. 17.5, *CurrSegs* is the set {wait(line2), wait(line6)}, so the expression will be added to both segments.

If *CurrStmt* is a boundary statement (e.g., `wait`), a new segment is added to *CurrSegs* with corresponding transition edges (lines 2–7 in Algorithm 5). Compound statements are processed by recursively iterating over the enclosed statements (lines 8–12), and conditional statements are processed recursively for each possible flow of control (from line 13). For example, the `break` and `continue` statements represent an unconditional jump in the program. For handling these keywords, the segments in *CurrSegs* move into the associated set *BreakSegs* or *ContSegs*, respectively. After completing the corresponding loop or `switch` statement, the segments in *BreakSegs* or *ContSegs* will be moved back to the *CurrSegs* set.

Algorithm 5 Segment graph generation

```
 1: function BUILDSG(CurrStmt, CurrSegs, BreakSegs, ContSegs)
 2:     if isBoundary(CurrStmt) then
 3:         NewSeg ← new segment
 4:         for Seg ∈ CurrSegs do
 5:             AddEdge(Seg, NewSeg)
 6:         end for
 7:         return CurrSegs ∪ { NewSeg }
 8:     else if isCompoundStmt(CurrStmt) then
 9:         for Stmt ∈ CurrStmt do
10:             CurrSegs ← BUILDSG(Stmt, CurrSegs, BreakSegs, ContSegs)
11:         end for
12:         return CurrSegs
13:     else if isIfStmt(CurrStmt) then
14:         AddExpression(IfCondition, CurrSegs);
15:         NewSegSet1 ← BUILDSG(IfBody, CurrSegs, BreakSegs, ContSegs)
16:         NewSegSet2 ← BUILDSG(ElseBody, CurrSegs, BreakSegs, ContSegs)
17:         return NewSegSet1 ∪ NewSegSet2
18:     else if isBreakStmt(CurrStmt) then
19:         BreakSegs ← BreakSegs ∪ CurrSegs
20:         CurrSegs ← ∅
21:         return CurrSegs
22:     else if isContinueStmt(CurrStmt) then
23:         ContSegs ← ContSegs ∪ CurrSegs
24:         CurrSegs ← ∅
25:         return CurrSegs
26:     else if isExpression(CurrStmt) then
27:         if isFunctionCall(CurrStmt) then
28:             return AddFunctionCall(CurrStmt, CurrSegs)        ▷ See Fig. 17.6a
29:         else
30:             AddExpression(CurrStmt, CurrSegs)
31:             return CurrSegs
32:         end if
33:     else if isLoop(CurrStmt) then
34:         return AddLoop(CurrStmt, CurrSegs)                    ▷ See Fig. 17.6b
35:     end if
36: end function
```

For brevity, we illustrate the processing of function calls and loops in Fig. 17.6. The analysis of function calls is shown in Fig. 17.6a. In step 1 the dashed circle represents the segment set *CurrSegs*. The RISC algorithm detects the function call expression and checks if the function is already analyzed. If not and it is encountered for the first time, the function is analyzed separately, as shown in step 2. Otherwise, the algorithm reuses the cached SG for the particular function. Then in step 3, each expression in segment 1 of the function is joined with each individual segment in *CurrSegs* (set 0). Finally, segments 4 and 5 represent the new set *CurrSegs*.

Correspondingly, Fig. 17.6b illustrates the SG analysis for a while loop. Again the dashed circle in step 1 represents the incoming set *CurrSegs*. The algorithm detects the `while` statement and analyzes the loop body separately. The graph for

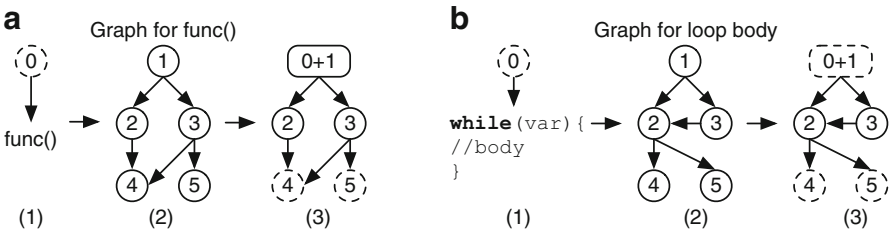


Fig. 17.6 Segment graph generation for functions and loops. (**a**) Function call processing. (**b**) Loop processing

the body of the loop is shown in step 2. Then each expression in segment 1 is joined into the segment set 0, and the new set *CurrSegs* becomes the joined set of 0+1, 4, and 5. Note that we have to consider set 0+1 for the case that the loop is not taken.

17.5.3 Static Conflict Analysis

The segment graph data structure serves as the foundation for static (compile time) conflict analysis. As outlined earlier, the OOO PDES scheduler must ensure that every new running thread is conflict-free with respect to any other threads in the READY and RUN queues. For this, we utilize the RISC compiler to detect any possible conflicts already at compile time.

Potential conflicts in SystemC include data hazards, event hazards, and timing hazards, all of which may exist among the current segments executed by the threads considered for parallel execution [6].

17.5.3.1 Data Hazards

Data hazards are caused by parallel or out-of-order accesses to shared variables. Three cases exist, namely, read after write (RAW), write after read (WAR), and write after write (WAW).

In the example in Fig. 17.7, if the simulator would issue the threads th_1 and th_2 in parallel, this would create a race condition, making the final value of s nondeterministic. Thus, the scheduler must not run th_1 and th_2 out of order. Note, however, that th_1 and th_2 can run in parallel in the segment after their second `wait` statement if the functions `f()` and `g()` are independent.

Since, data hazards stem from the code in specific segments, RISC analyzes data conflicts statically at compile time and creates a table where the scheduler can then at run time quickly look up any potential conflicts between active segments.

Formally, we define a data conflict table $CT[N, N]$ where N is the total number of segments in the application model: $CT[i, j] = true$, iff there is a potential data conflict between the segments seg_i and seg_j; otherwise, $CT[i, j] = false$.

To build the conflict table, the compiler generates for each segment a variable access list which contains all variables accessed in the segment. Each entry is a

Fig. 17.7 Example of WAW conflict: Two parallel threads th_1 and th_2 start at the same time but write to the shared variable s at different times. Simulation semantics require that th_1 executes first and sets s to 0 at time $(5, 0)$, followed by th_2 setting s to its final value 1 at time $(10, 0)$

```
1  int s;
2
3  thread1()
4  { wait( 5, SC_MS);
5    s = 0;
6    wait(10, SC_MS);
7    f();
8  }
9
10 thread2()
11 { wait(10, SC_MS);
12   s = 1;
13   wait(10, SC_MS);
14   g();
15 }
```

tuple (*Symbol*, *AccessType*) where *Symbol* is the variable and *AccessType* specifies read only (R), write only (W), read write (RW), or pointer access (Ptr).

Finally, the compiler produces the conflict table $CT[N, N]$ by comparing the access lists for each segment pair. If two segments seg_i and seg_j share any variable with access type (W) or (RW), or there is any pointer access by seg_i or seg_j, then this is marked as a potential conflict.

Figure 17.8 shows an example SystemC model where two threads th_1 and th_2 which run in parallel in modules *M1* and *M2*, respectively. Both threads write to the global variable x, th_1 in lines 14 and 27, and th_2 in line 35 since reference p is mapped to x. Before we can mark these WAW conflicts in the data conflict table, we need to generate the segment graph for this example. The SG with corresponding source code lines is shown in Fig. 17.9a, whereas Fig. 17.9b shows the variable accesses by the segments. Note that segments 3 and 4 of thread th_1 write to x, as well as segment 8 of th_2 which writes to x via the reference p. Thus, segments 3, 4, and 8 have a WAW conflict. This is marked properly in the corresponding data conflict table shown in Fig. 17.10a.

In general, not all variables are straightforward to analyze statically. SystemC models can contain variables at different scopes, as well as ports which are connected by port maps. The RISC compiler distinguishes and supports the following cases for the static variable access analysis.

1. Global variables, e.g., x, y in lines 2 and 3 of Fig. 17.8: This case is discussed above and is handled directly as tuple (*Symbol*, *AccessType*).
2. Local variables, e.g., *temp* in line 10 for Module $M1$: Local variables are stored on the stack and cannot be shared between different threads. Thus, they can be ignored in the variable access analysis.
3. Instance member variables, e.g., i in line 2 for Module $M2$: Since classes can be instantiated multiple times and then their variables are different, we need to distinguish them by their complete instance path added to the variable name. For example, the actual symbol used for the instance variable i in module $M2$ is $m.m2.i$.

```
#include "systemc.h"
int x = 0;
int y;
SC_MODULE(M1) {                          // Module M1
  SC_HAS_PROCESS(M1);
  sc_event &event;
  M1(sc_module_name name, sc_event &e): event(e)
  { SC_THREAD(main); }
  void main() {
    int temp = 0;
    while(temp++<2) {
      wait(1, SC_MS);
      wait(event);
      x = temp;
    }
    wait(3, SC_MS);
    x = 27;
  }
};
SC_MODULE(M2) {                          // Module M2
  SC_HAS_PROCESS(M2);
  int i;
  int &p;
  sc_event &event;
  M2(sc_module_name name, int &pp, sc_event &e):
    sc_module(name), p(pp), i(0), event(e)
  { SC_THREAD(main); }
  void main() {
    do {
      wait(2, SC_MS);
      y = i;
      event.notify(SC_ZERO_TIME);
    } while(i++<2);
    wait(4, SC_MS);
    p = 42;
  }
};
SC_MODULE(Main) {                        // Module Main
  sc_event event;
  M1 m1;
  M2 m2;
  Main(sc_module_name name):
    sc_module(name), m1("m1", event), m2("m2", x, event)
  { }
};
int sc_main(int argc, char **argv) {
  Main m("main");
  sc_start();
  return 0;
}
```

Fig. 17.8 SystemC example with two parallel threads in modules *M1* and *M2*

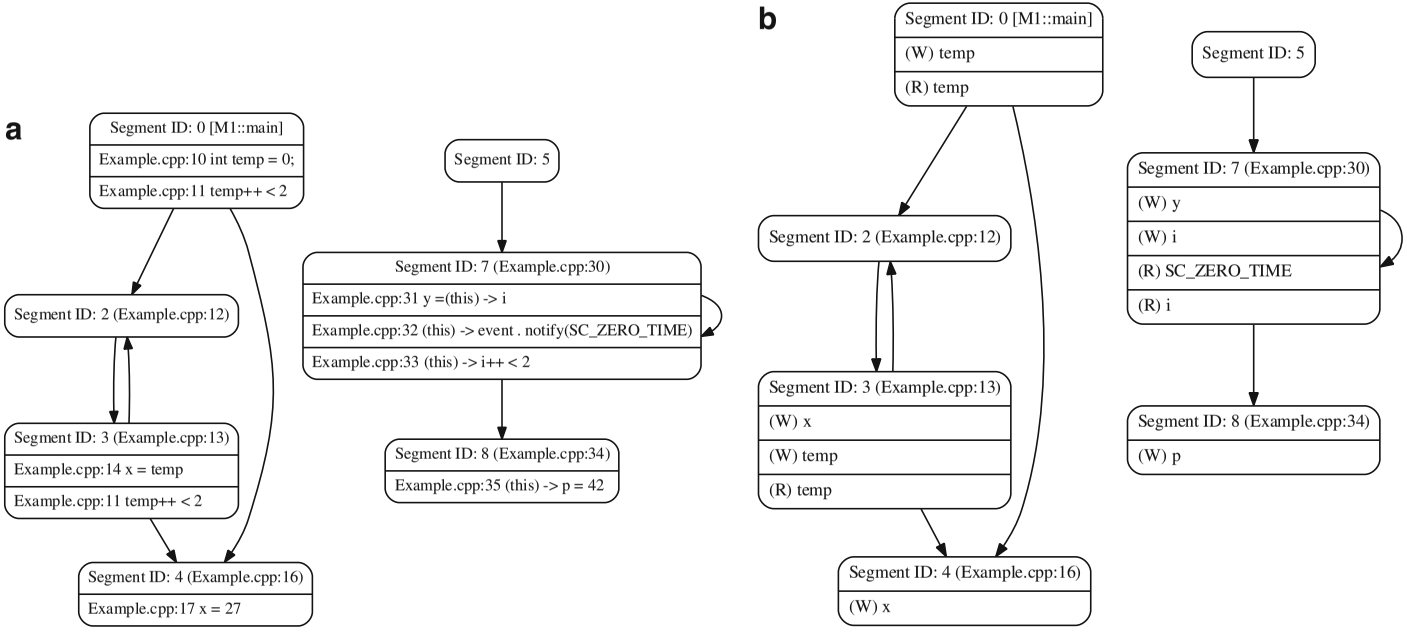


Fig. 17.9 Segment graphs generated by RISC for the example in Fig. 17.8. (**a**) Source code SG for Fig. 17.8. (**b**) Variable access SG for Fig. 17.8

a

	0	2	3	4	5	7	8
0							
2							
3			T	T			T
4			T	T			T
5							
7						T	
8			T	T			T

b

	0	2	3	4	5	7	8
0							
2							
3							
4							
5							
7			T				
8							

Fig. 17.10 Data conflict and event notification tables for the example in Fig. 17.8. (**a**) Data conflict table. (**b**) Event notification table

4. References, e.g., p in line 23 in module $M2$: RISC follows references through the module hierarchy and determines the actual mapped target variable. For instance, p is mapped to the global variable x via the mapping in line 43.
5. Pointers: RISC currently does not perform pointer analysis. This is planned as future work. For now, RISC conservatively marks all segments with pointer accesses as potential conflict with all other segments.

17.5.3.2 Event Hazards

Thread synchronization through event notification also poses hazards to out-of-order execution. Specifically, thread segments are dependent when one is waiting for an event notified by another.

We define an event notification table $NT[N, N]$ where N is the total number of segments: $NT[i, j] = true$, iff segment seg_i notifies an event that seg_j is waiting for; otherwise, $NT[i, j] = false$. Note that in contrast to the data conflict table, the event notification table is not symmetric.

Figure 17.10b shows the event notification table for the SystemC example in Fig. 17.8. For instance, $NT[7, 3] = true$ since segment 7 notifies the event e in line 32, which segment 3 is waiting for in line 13.

Note that in order to identify event instances properly, RISC uses the same scope and port map handling for events as described above for data variables.

17.5.3.3 Timing Hazards

The local time for an individual thread in OOO PDES can pose a timing hazard when the thread runs too far ahead of others. To prevent this, we analyze the minimum time advances of threads at segment boundaries. For SystemC, there are three cases with different time increments, as listed in Table 17.1.

In order for the scheduler to avoid timing hazards, we let the compiler prepare two time advance tables, one for the segment a thread is currently in and one for the next segment(s) that a thread can reach in the following scheduling step.

The current time advance table $CTime[N]$ lists the time increment that a thread will experience when it enters the given segment. For the SystemC example in Figs. 17.8 and 17.11a shows the corresponding current time advance table. Here

Table 17.1 Time advances at `wait` segment boundaries

Segment boundary	Time increment		Add to (t', δ')
`wait`(t)	Increment by time t	$(t:0)$	$(t'+t, 0)$
`wait`(event)	Increment by one delta count	$(0:1)$	$(t', \delta'+1)$
`wait`(immediate event)	No increment	$(0:0)$	(t', δ')

a

0	2	3	4	5	7	8
(0:0)	(1:0)	(0:0)	(3:0)	(0:0)	(2:0)	(4:0)

b

0	2	3	4	5	7	8
(1:0)	(0:0)	(1:0)	∞	(2:0)	(2:0)	∞

Fig. 17.11 Current and next time advance tables for the example in Fig. 17.8. (**a**) Current time advance table. (**b**) Next time advance table

Table 17.2 Examples for direct and indirect timing hazards

Situation	th_1	th_2	Hazard?
Direct timing hazard	$(10:2)$	$(10:0)$, next segment at $(10:1)$	Yes
	$(10:2)$	$(10:0)$, next segment at $(12:0)$	No
Indirect timing hazard	$(10:2)$	$(10:0)$, wakes th_3 at $(10:1)$	Yes
	$(10:2)$	$(10:1)$, wakes th_3 at $(10:2)$	No

for instance, the `wait(2,SC_MS)` in line 30 at the beginning of segment 7 defines $CTime[7] = (2, 0)$.

On the other hand, the next time advance table $NTime[N]$ lists the time increment that a thread will incur when it leaves the given and enters the next segment. Since there may be more than one next segment, we list in the table the minimum of the time advances, which is the earliest time the thread can become active again. Formally: $NTime[i] = min\{CTime[j], \forall seg_j$ which follow $seg_i\}$.

For example, Fig. 17.11b lists $NTime[0] = (1, 0)$ since segment 0 is followed by segment 2 with increment $(1, 0)$ and segment 4 with increment ∞.

There are two types of timing hazards, namely, direct and indirect ones. For a candidate thread th_1 to be issued, a direct timing hazard exists when another thread th_2, that is safe to run, resumes its execution at a time earlier than the local time of th_1. In this case, the future of th_2 is unknown and could potentially affect th_1. Thus, it is not safe to issue th_1.

Table 17.2 shows an example where a thread th_1 is considered for execution at time $(10:2)$. If there is a thread th_2 with local time $(10:0)$ whose next segment runs at time $(10:1)$, i.e., before th_1, then the execution of th_1 is not safe. However, if we know from the time advance tables that th_2 will resume its execution later at $(12:0)$, no timing hazard exists with respect to th_2.

An indirect timing hazard exists, if a third thread th_3 can wake up earlier than th_1 due to an event notified by th_2. Again, Table 17.2 shows an example. If th_2 at $time(10:0)$ potentially wakes a thread th_3 so that th_3 runs in the next delta cycle $(10:1)$, i.e., earlier than th_1, then it is not safe to issue th_1.

17.5.4 Source Code Instrumentation

As shown above in Fig. 17.4 on page 547, the RISC compiler and parallel simulator work closely together. The compiler performs the complex conservative static analysis and passes the analysis results to the simulator which then can make safe scheduling decisions quickly.

More specifically, the RISC compiler passes all the generated conflict tables to the simulator, namely, the data conflict table, the event notification table, as well as the current and next time advance tables. In addition, the compiler instruments the model source code so that the simulator can properly identify each thread and each segment by unique numeric IDs.

To pass information from the compiler to the simulator, RISC uses automatic model instrumentation. That is, the intermediate model generated by the compiler contains instrumented (automatically generated) source code which the simulator then can rely on. At the same time, the RISC compiler also instruments user-defined SystemC channels with automatic protection against race conditions among communicating threads. The source code instrumentation with segment IDs, conflict tables, and automatic channel protection is a part of model "recoding" (i.e., the "R" in RISC). For the future, we envision additional recoding tasks performed by RISC, such as model transformation, optimization, and refinement.

In total, the RISC source code instrumentation includes four major components:

1. Segment and instance IDs: Individual threads are uniquely identified by a creator instance ID and their current code location (segment ID). Both IDs are passed into the simulator kernel as additional arguments to all scheduler entry functions, including `wait` calls and thread creation.
2. Data and event conflict tables: Segment concurrency hazards due to potential data conflicts, event conflicts, or timing conflicts are provided to the simulator as two-dimensional tables indexed by a segment ID and instance ID pair. For efficiency, these table entries are filtered for scope, instance path, and reference and port mappings.
3. Current and next time advance tables: The simulator can make better scheduling decisions by looking ahead in time if it can predict the possible future thread states. This possible optimization is discussed in detail in [5] but remains as a future work item for the current RISC prototype.
4. User-defined channel protection: SystemC allows the user to design channels for custom inter-thread communication. To ensure that such user-defined communication remains safe also in the OOO PDES situation where threads execute truly in parallel and out of order, the RISC compiler automatically inserts locks (binary semaphores) into these user-defined channel instances (which are acquired at entry and released upon leaving) so that mutually exclusive execution of the channel methods is guaranteed. Otherwise, race conditions could exist when communicating threads exchange data.

After this automatic source code instrumentation, the RISC compiler passes the generated intermediate model to the underlying regular C++ compiler which

produces the final simulator executable by linking the instrumented code against the RISC extended SystemC library.

17.6 Experimental Evaluation

We now present two SystemC application models as examples and evaluate the performance of DES, PDES, and OOO PDES algorithms on modern multi-core hosts. As DES representative and baseline reference, we will use the open-source proof-of-concept library [16] provided by the SystemC Language Working Group [31] of the Accellera Systems Initiative [1]. As OOO PDES representative, we will use the RISC [21] which is also available as open source [20]. For synchronous PDES, we will use an in-house version of RISC where the out-of-order scheduling features are disabled. To ensure a fair comparison, all simulator packages are based on Posix threads and compiled with the same optimization settings, and of course run on the same host environment.

17.6.1 Conceptual DVD Player Example

Our first example is an abstract model of a DVD player, as shown in Fig. 17.12. While this SystemC model is conceptual only, it is well-motivated and very educational as it clearly demonstrates the differences between the DES, PDES, and OOO PDES algorithms.

As listed in Fig. 17.12, the SystemC modules representing the video and audio decoders operate in an infinite loop, reading a frame from the input stream, decoding it, and sending it out to the corresponding monitor modules. Since the video and audio frames are data independent, the decoders run in parallel and

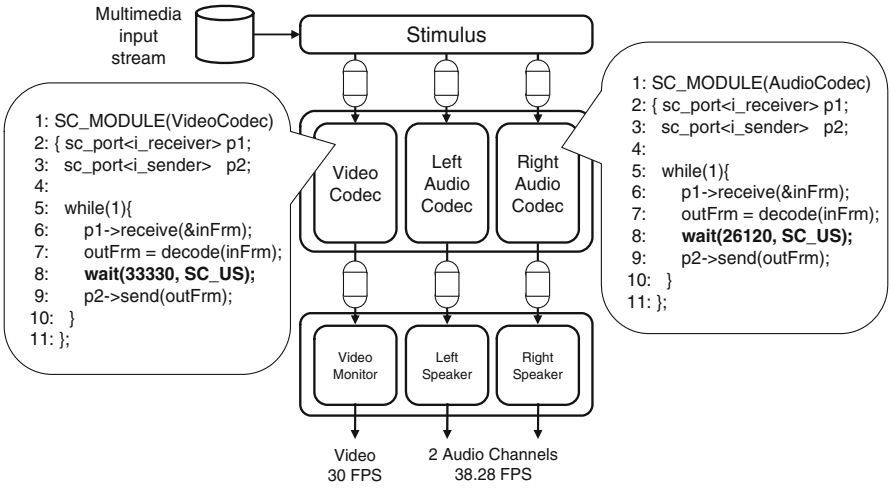


Fig. 17.12 Conceptual DVD player example with a video and two audio stream decoders

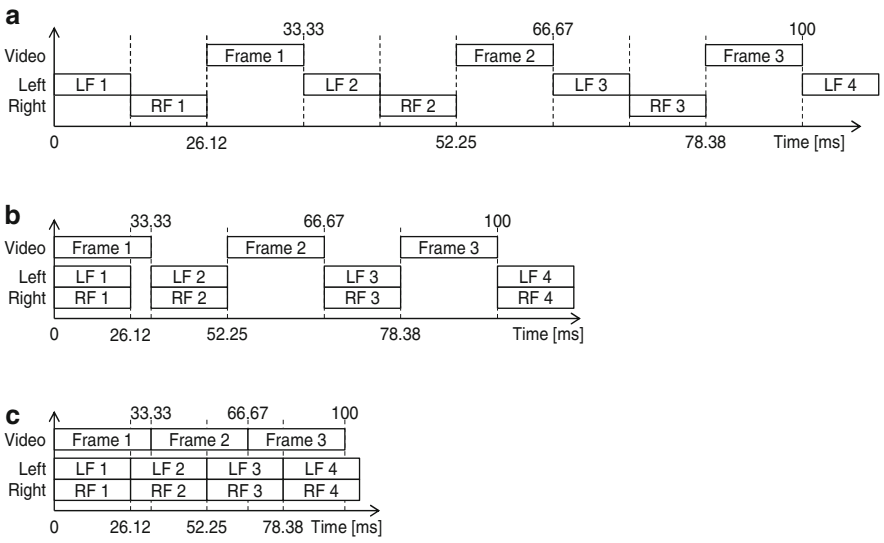


Fig. 17.13 Time lines for the DVD player example during simulation. (**a**) DES schedule: sequential execution, only one task at any time. (**b**) Synchronous PDES schedule: parallel execution only at the same simulation times. (**c**) Out-of-order PDES schedule: fully parallel execution (the same as in reality!)

output the decoded frames according to their channel rate, 30 frames per second video (delay 33.33 ms) and 38.28 frames per second audio (delay 26.12 ms), respectively.

Figure 17.13 depicts the time lines of simulating the DVD player according to DES, PDES, and OOO PDES semantics. As expected, the DES schedule in Fig. 17.13a executes only a single task at all times. Synchronous PDES in Fig. 17.13b is able to parallelize the decoders for the left and right audio channels but cannot exploit parallelism for decoding the video channel due to its different frame rate. Only the OOO PDES schedule in Fig. 17.13c shows the fully parallel execution that we also expect in reality. Note that the artificially discretized timing in the model prevents PDES from full parallelization. In contrast, OOO PDES with thread-local timing achieves the goal.

Our experimental measurements listed in Table 17.3 confirm the analysis of Fig. 17.13. For both experiments, the synchronous PDES gains about 50% simulation speed over the reference DES. However, the out-of-order PDES beats the synchronous approach by another 100% improvement.

17.6.2 Mandelbrot Renderer Example

As a representative example of very computation intensive and highly parallel applications, we have evaluated the three-simulation algorithms also on a graphics pipeline model that renders a sequence of images of the Mandelbrot set.

Table 17.3 Experimental results for the DVD player example. RISC V0.2.1 simulator performance (Posix-thread based) on a 8-core Intel® Xeon® host PC (E3-1240 CPU, 4 cores, 2 hyper-threads) at 3.4 GHz

Movie	Simulator	DES	PDES	OOO PDES
10 second stream	Run time	6.98 s	4.67 s	2.94 s
	CPU load	97%	145%	238%
	Speedup	1×	1.49×	2.37×
100 second stream	Run time	68.21 s	45.91 s	28.13 s
	CPU load	100%	149%	251%
	Speedup	1×	1.49×	2.42×

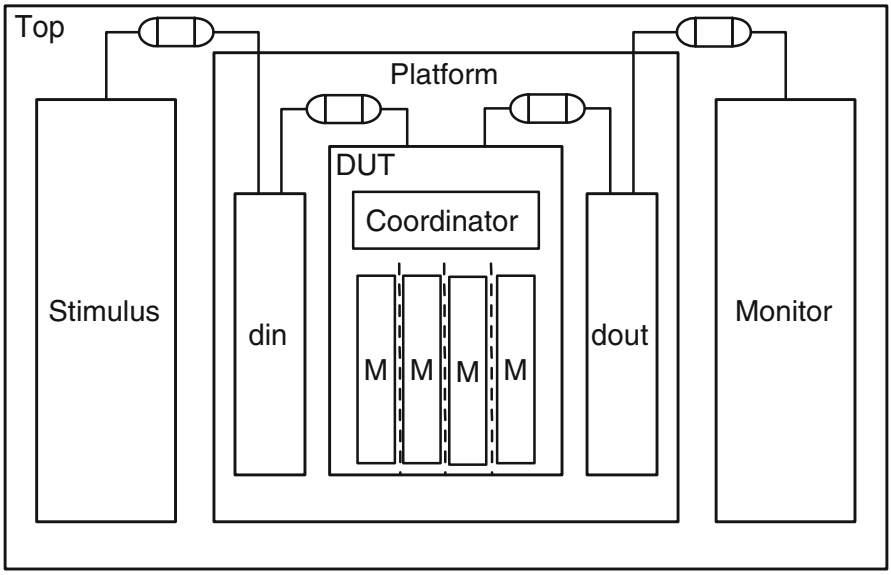


Fig. 17.14 Mandelbrot renderer example: Stimulus generates target coordinates in the complex plane for which the DUT renders the corresponding Mandelbrot image and sends it to the Monitor module. In the DUT, a Coordinator distributes slices of the target coordinates to parallel Mandelbrot worker threads and synchronizes their progress

Figure 17.14 shows the block diagram of our Mandelbrot renderer model in SystemC.

The number of Mandelbrot worker threads is user configurable as an exponent of 2, for example, 4 as illustrated in Fig. 17.14. Each worker thread computes a different horizontal slice of the image and works independently and in parallel to the others. If enabled in the SystemC model, the progress of the workers' computation is displayed in a window, as shown in Fig. 17.15.

Table 17.4 shows the measured experimental results for the Mandelbrot renderer with different numbers of worker threads, one per image slice, as indicated in the first column. Since the example is "embarrassingly parallel" in the DUT and otherwise contains only comparatively little sequential computation, both

Fig. 17.15 Screenshot of the Mandelbrot set renderer in action: The progress of the parallel threads, which collaboratively compute a Mandelbrot set image, can be viewed live in a window on screen. Here, eight SystemC threads compute the eight horizontal slices of the image in parallel. Note that this visualization clearly shows the difference between the sequential DES simulation (which computes only one slice at any time) and the parallel PDES algorithms (which are shown here)

Table 17.4 Experimental results for the Mandelbrot renderer example. RISC V0.2.1 simulator performance (Posix-thread based) on a 32-core Intel® Xeon® host PC (2 E5-2680 CPUs, 8 cores, 2 hyper-threads) at 2.7 GHz

	DES		PDES			OOO PDES		
Slices	Run time	Load	Run time	Load	Speedup	Run time	Load	Speedup
1	162.13 s	99%	162.06 s	100%	1.0×	161.90 s	100%	1.0×
2	162.19 s	99%	96.50 s	168%	1.7×	96.48 s	168%	1.7×
4	162.56 s	99%	54.00 s	305%	3.0×	53.85 s	304%	3.0×
8	163.10 s	99%	29.89 s	592%	5.5×	30.05 s	589%	5.4×
16	164.01 s	99%	19.03 s	1050%	8.6×	20.08 s	997%	8.2×
32	165.89 s	99%	11.78 s	2082%	14.1×	11.99 s	2023%	13.8×
64	170.32 s	99%	9.79 s	2607%	17.4×	9.85 s	2608%	17.3×
128	174.55 s	99%	9.34 s	2793%	18.7×	9.39 s	2787%	18.6×
256	185.47 s	100%	8.91 s	2958%	20.8×	8.90 s	2964%	20.8×

parallel simulators respond with impressive performance speedups, more than 20× compared to the Accellera reference simulator. With growing parallelism, the simulation speed increases almost linearly with respect to the number of parallel workers, up until to the point where the number of software threads reaches the number of available hardware threads (2 CPUs, 8 cores with 2 hyper-threads each, so 32 in total).

We can also observe that, for this example, the synchronous PDES and the OOO PDES perform the same since the differences are within the noise range of the measurement accuracy. This matches the expectation, because the Mandelbrot workers are all synchronized (locked in) due to their communication with the coordinator thread and thus out-of-order execution cannot be exploited here.

As for scalability, PDES and OOO PDES approaches scale very well, if we base our expectation on the host hardware capabilities and the amount of parallelism

exposed in the application model (which is the fundamental limitation of any PDES). Overall, we observe that parallel simulation has the potential to improve simulation speed by an order of magnitude or more. In this book chapter we do not evaluate the overhead of the static analysis incurred at compile time because our current compiler implementation does not produce meaningful results due to its unoptimized ROSE foundation. Generally, compile time for OOO PDES increases moderately, but is amortized by the typically longer and more frequent simulations [6].

17.7 Conclusion

In the era of stagnant processor clock frequencies and the growing availability of inexpensive multi- and many-core architectures, parallel simulation is a must-have for the efficient validation and exploration of embedded system-level design models. Consequently, the traditional purely sequential DES approach, as provided by the open-source SystemC reference simulator, is inadequate for the future when the system complexity keeps growing at its current exponential pace.

In this chapter, we have reviewed the classic discrete event-based simulation techniques with a focus on state-of-the-art parallel solutions (synchronous and out-of-order PDES) that are particularly suited for the de facto and official IEEE standard SystemC. While many approaches have been proposed in the research community, we have detailed the OOO PDES approach [6] being developed in the Recoding Infrastructure for SystemC (RISC) [21]. The open-source RISC project provides a dedicated SystemC compiler and corresponding out-of-order parallel simulator as proof-of-concept implementation to the research community and industry.

The OOO PDES technology stands out from other approaches as an aggressive yet conservative modern simulation approach beyond traditional PDES, because it can exploit parallel computing resources to the maximum extend and thus achieves fastest simulation speed. At the same time, it can preserve compliance with traditional SystemC semantics and support legacy models without modification or loss of accuracy.

Acknowledgments This work has been supported in part by substantial funding from Intel Corporation. The authors thank Intel Corporation for the valuable support and fruitful collaboration. The authors also thank the anonymous reviewers for valuable suggestions to improve this chapter.

References

1. Accellera Systems Initiative. http://www.accellera.org
2. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the spring joint computer conference, AFIPS'67 (Spring), 18–20 Apr 1967. ACM, New York, pp 483–485. doi:10.1145/1465482.1465560

3. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of the international conference on hardware/software codesign and system synthesis, Newport Beach
4. Chandy K, Misra J (1979) Distributed simulation: a case study in design and verification of distributed programs. IEEE Trans Softw Eng SE-5(5):440–452
5. Chen W, Dömer R (2013) Optimized out-of-order parallel discrete event simulation using predictions. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE)
6. Chen W, Han X, Chang CW, Liu G, Dömer R (2014) Out-of-order parallel discrete event simulation for transaction level models. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 33(12):1859–1872. doi:10.1109/TCAD.2014.2356469
7. Chen W, Han X, Dömer R (2011) Multi-core simulation of transaction level models using the system-on-chip environment. IEEE Des Test Comput 28(3):20–31
8. Chen W, Han X, Dömer R (2012) Out-of-order parallel simulation for ESL design. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE)
9. Dömer R, Chen W, Han X, Gerstlauer A (2011) Multi-core parallel simulation of system-level description languages. In: Proceedings of design automation conference. Asia and South Pacific (ASPDAC), pp 311–316
10. Dömer R, Gerstlauer A, Gajski D (2002) SpecC language reference manual, version 2.0. SpecC technology open consortium. http://www.specc.org
11. Ezudheen P, Chandran P, Chandra J, Simon BP, Ravi D (2009) Parallelizing SystemC kernel for fast hardware simulation on SMP machines. In: PADS'09: proceedings of the 2009 ACM/IEEE/SCS 23rd workshop on principles of advanced and distributed simulation, pp 80–87
12. Fujimoto R (1990) Parallel discrete event simulation. Commun ACM 33(10):30–53
13. Gajski DD, Zhu J, Dömer R, Gerstlauer A, Zhao S (2000) SpecC: specification language and design methodology. Kluwer Academic Publishers, Boston
14. Gerstlauer A (2010) Host-compiled simulation of multi-core platforms. In: Proceedings of the international symposium on rapid system prototyping (RSP), Washington, DC
15. Grötker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic Publishers, Dordrecht
16. Group SLW SystemC 2.3.1, core SystemC language and examples. http://accellera.org/downloads/standards/systemc
17. Huang K, Bacivarov I, Hugelshofer F, Thiele L (2008) Scalably distributed SystemC simulation for embedded applications. In: International symposium on industrial embedded systems, SIES 2008, pp 271–274
18. IEEE Computer Society (2011) IEEE standard 1666-2011 for standard SystemC language reference manual. IEEE, New York
19. Kaushik A, Patel HD (2013) SystemC-clang: an open-source framework for analyzing mixed-abstraction SystemC models. In: Proceedings of the forum on specification and design languages (FDL), Paris
20. Liu G, Schmidt T, Doemer R Recoding infrastructure for SystemC (RISC) compiler and simulator. http://www.cecs.uci.edu/~doemer/risc.html
21. Liu G, Schmidt T, Dömer R (2015) RISC compiler and simulator, alpha release V0.2.1: out-of-order parallel simulatable SystemC subset. Technical Report CECS-TR-15-02, Center for Embedded and Cyber-physical Systems, University of California, Irvine
22. Mukherjee S, Reinhardt S, Falsafi B, Litzkow M, Hill M, Wood D, Huss-Lederman S, Larus J (2000) Wisconsin wind tunnel II: a fast, portable parallel architecture simulator. IEEE Concurr 8(4):12–20
23. Nanjundappa M, Patel HD, Jose BA, Shukla SK (2010) SCGPSim: a fast SystemC simulator on GPUs. In: Proceedings of design automation conference. Asia and South Pacific (ASPDAC)
24. Nicol D, Heidelberger P (1996) Parallel execution for serial simulators. ACM Trans Model Comput Simul 6(3):210–242

25. Quinlan DJ (2000) ROSE: compiler support for object-oriented frameworks. Parallel Process Lett 10(2/3):215–226
26. Schmidt T, Liu G, Dömer R (2016) Automatic generation of thread communication graphs from SystemC source code. In: Proceedings of international workshop on software and compilers for embedded systems (SCOPES)
27. Schumacher C, Leupers R, Petras D, Hoffmann A (2010) parSC: synchronous parallel SystemC simulation on multi-core host architectures. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 241–246
28. Sinha R, Prakash A, Patel HD (2012) Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs. In: Proceedings of design automation conference. Asia and South Pacific (ASPDAC)
29. Sirowy S, Huang C, Vahid F (2010) Online SystemC emulation acceleration. In: Proceedings of design automation conference (DAC)
30. Stattelmann S, Bringmann O, Rosenstiel W (2011) Fast and accurate source-level simulation of software timing considering complex code optimizations. In: Proceedings of design automation conference (DAC)
31. SystemC Language Working Group (LWG). http://accellera.org/activities/working-groups/systemc-language
32. SystemC TLM-2.0. http://www.accellera.org/downloads/standards/systemc/tlm
33. Weinstock J, Schumacher C, Leupers R, Ascheid G, Tosoratto L (2014) Time-decoupled parallel SystemC simulation. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Dresden
34. Yun D, Kim S, Ha S (2012) A parallel simulation technique for multicore embedded systems and its performance analysis. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 31(1):121–131
35. Zhu J, Dömer R, Gajski DD (1997) Syntax and semantics of the SpecC language. In: International symposium on system synthesis (ISSS), Osaka

Multiprocessor System-on-Chip Prototyping Using Dynamic Binary Translation

18

Frédéric Pétrot, Luc Michel, and Clément Deschamps

Abstract

Dynamic binary translation is a processor emulation technology that allows to execute in a very efficient manner a binary program for an instruction-set architecture A on a processor having instruction-set architecture B. This chapter starts by giving a rapid overview of the dynamic binary translation process and its peculiarities. Then, it focuses on the support for SIMD instruction and the translation for VLIW architectures, which bring upfront new challenges for this technology. Next, it shows how the translation process can be enhanced by the insertion of instructions to monitor nonfunctional metrics, with the aim of giving, for instance, timing or power consumption estimations. Finally, it details how it can be integrated within virtual prototyping platforms, looking in particular at the synchronization issues.

Acronyms

DBT	Dynamic Binary Translation
ILP	Instruction-Level Parallelism
ISA	Instruction-Set Architecture
ISS	Instruction-Set Simulator
MMU	Memory Management Unit
MPSoC	Multi-Processor System-on-Chip
OS	Operating System
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multi-Processing
SSA	Static Single Assignment

F. Pétrot (✉)
Université de Grenoble Alpes, Grenoble, France
e-mail: frederic.petrot@univ-grenoble-alpes.fr

L. Michel • C. Deschamps
Antfield SAS, Grenoble, France
e-mail: luc.michel@antfield.fr; clement.deschamps@antfield.fr

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_20

TB	Translation Block
TLM	Transaction-Level Model
VLIW	Very Long Instruction Word
VP	Virtual Prototype
WAR	Write-After-Read

Contents

18.1 Introduction

Virtual Prototype (VP) serve different purposes, and depending on these purposes, the acceptable "accuracy *vs* speed of simulation" trade-off is very different. More than two decades ago, Transaction-Level Model (TLM) was introduced as a way to abstract time and data and clearly decouple computations from communications. Compared to Register Transfer Level (RTL) which focuses on implementation and targets on accuracy, TLM aims at giving a high-level view of the system so that it is possible to quickly take design decisions. What has been intuited at that time is now recognized as an actual solution for early software development and design space exploration of hardware/software systems [5]. The system-on-chip industry has seen the value of having models sitting in between RTL and fully analytical formulas and has adopted this kind of modeling strategy quite rapidly [15].

In the context of an ever-increasing number of programmable cores in integrated devices, the simulation of the software part of a system becomes a critical issue. Even though TLM is well suited for hardware design, it says nothing about the way the software that runs on the hardware part of the system should be executed. Several strategies can be thought of, ranging from interpretive instruction-set simulation of the cross-compiled target binary code to the native execution of host compiled code using the Operating System (OS) calls as simulation callbacks [24]. Figure 18.1 summarizes the main strategies used for software execution on top of transaction-level hardware.

The three first software execution approaches can be classified as interpretive ones.

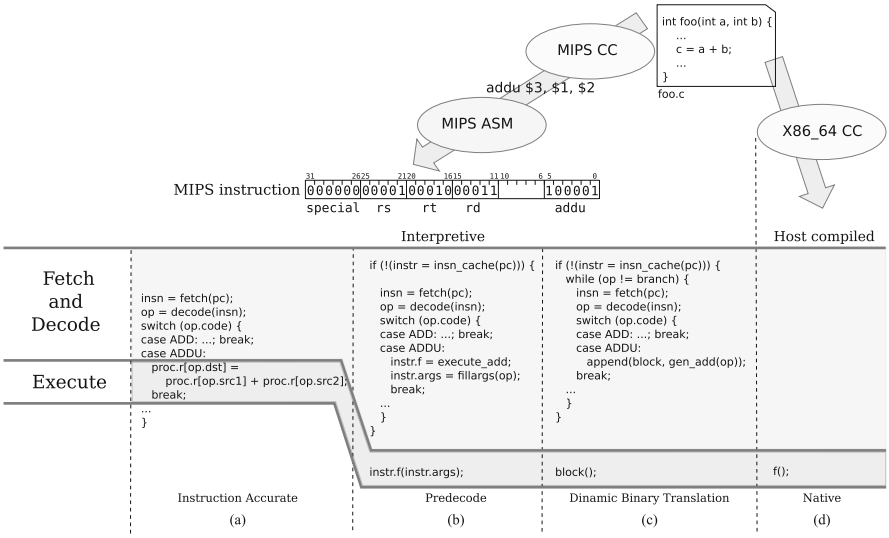


Fig. 18.1 Major software execution strategies in TLM environments

Instruction accurate interpretation (Fig. 18.1a) is the textbook method for executing cross-compiled code. Each instruction is read from memory, decoded, and executed. Decoding can be done using a big `switch`-like control structure or a functions pointer table [4].

Predecode (Fig. 18.1b) is based on the same principle, but the instructions are fetched and decoded, and placed in a cache in which they are identified by their program counter (`pc`) [22]. Then, if the `pc` matches a cache entry, the instruction is directly executed through a function call. Using a decode cache brings up new issues as it must be kept up to date when some code region is modified. Some applications heavily rely on dynamic compilation, so it is necessary to handle it with efficiency. Incidentally, this cache can be the actual simulation model of the processor instruction cache, as the high hit rate it usually has ensures that few fetch/decode will be redone without necessity. Furthermore, provided the simulated instruction caches ensure coherency, multiprocessor systems work out of the box with this approach.

Dynamic Binary Translation (DBT) pushes the limits further by fetching and decoding an entire sequence of code ended by a branch at once, translating it into host code with identical behavior, and caching the result [11]. The whole sequence is then executed atomically, avoiding to check per instruction the presence of the `pc` in the cache. As a translation cache is used, the same issues as predecode arise, but then using the model of an instruction cache is not possible as there is no such thing as a single target instruction in DBT.

The last approach, introduced here only for completeness, takes a fully different angle, as the high-level code to be executed is directly compiled for the host. Many

different approaches have been proposed, as detailed in ▸ Chap. 19, "Host-Compiled Simulation" of this book. Anyhow, these native or host-compiled approaches need to access the hardware and, at some point, rely on an operating system or hardware abstraction layer API to implicitly let the simulator execute the hardware models. A clear limitation of these strategies is that it is hardly possible to handle self-modifying code or dynamic code generation; however, not all applications need it.

A current trend in system-level simulation is to use DBT to execute target code and TLM to model hardware [1, 16, 23]. Indeed, a desirable goal is to define a framework which provides a way to virtually prototype full hardware/software multiprocessor systems with an entire software stack, including the operating systems and the device drivers. As this requires to execute the cross-compiled binary code of all software layers at high speed, dynamic binary translation is the more suitable software execution technique.

On the one hand, modern DBT engines take their root in virtualization, an effort that took place in computer science to make possible the execution of several OS in isolation concurrently on the same processor [28], based on the *virtual machine* technology developed in the early 1960s [7].

On the other hand, due to the constraints of consumer system integration (form factor, packaging, power, heat, etc.), each application or class of application still calls for a specific circuit in order to fit into the performance/power budget. The Multi-Processor System-on-Chip (MPSoC) design approaches aim firstly at clearly separating computation from communication, using interfaces that are standardized, allowing to quickly exchange one IP, including processors, by another. Secondly, they aim at producing figures of merits, such as code sequence run times and interrupt latencies, used bandwidth on an interconnect, even energy or power information, depending of the architectural choices. Due to the increase in number of programmable cores and software in the near to come SoCs, the availability of virtual platforms providing a structural view of the system and fast application and OS code execution with a reliable accuracy is an important issue.

If modularity is of primary importance in MPSoC design, it is not the case for virtualization which targets a single one-shot hardware platform with an as high as possible execution speed. Although the goals of the hardware/software cosimulation and virtualization may seem very different, at the end of the day the way to achieve these different goals are similar.

The rest of the chapter is devoted to clarifying the necessary points to build an operational optimized MPSoC simulator which makes use of DBT as software execution engine.

18.2 Dynamic Binary Translation Basics

Full virtualization allows the execution of an operating system, called guest, on top of another operation system, called host, without any modification. One of the main issues in virtualization is the execution of the privileged guest operating

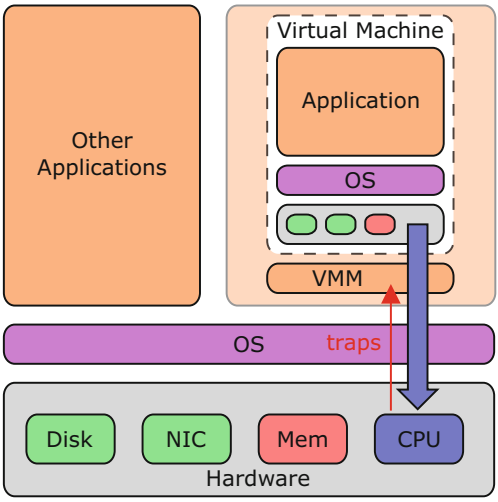


Fig. 18.2 Trap and emulate based virtualization

system instructions since the simulator in running on the host operating system as an unprivileged application. Another delicate issue is the interception of I/O operations of guest operating system. The classical solution for full virtualization, called trap-and-emulate, is presented Fig. 18.2. It assumes the direct execution of the simulated machine binary code on the host processor; therefore, the host processor and the target processor (the one on which the guest operating system is run) must be identical. This solution is based on a virtual machine monitor (VMM) which takes control whenever a trap caused by a privileged operation executed in the unprivileged context of the virtual machine occurs. However, not all processors are fully virtualizable [25], typically because some privileged instructions executed in user mode do not trap. To work around this problem, the binary translation approach using DBT, as depicted Fig. 18.3, can be used. This approach has no constraints concerning the host and target processor types, as all instructions of the guest operating system binary code, including the privileged ones, are replaced by unprivileged instructions and/or system calls.

Apart from virtualization, historically DBT has also been used for transparently running legacy software compiled for a processor on either a new version of this processor or an entirely different processor. Apple used this technology when transitioning from the PowerPC architecture to the x86 one (Rosetta by Transitive Technologies, now acquired by IBM). When Intel introduced the IA64, DBT from x86 code was also applied [2]. While the former case translates from scalar to scalar architectures, the latter one does a scalar to Very Long Instruction Word (VLIW) architecture translation, which requires a more complex process.

Figure 18.4 presents the general principle of binary-translation-based simulators. The simulator starts by verifying if the current `pc` of the simulated processor has already been encountered. If not, the *binary translation* stage begins.

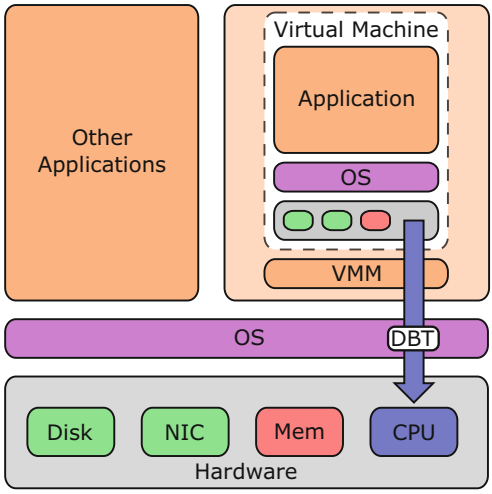


Fig. 18.3 Dynamic binary translation based virtualization

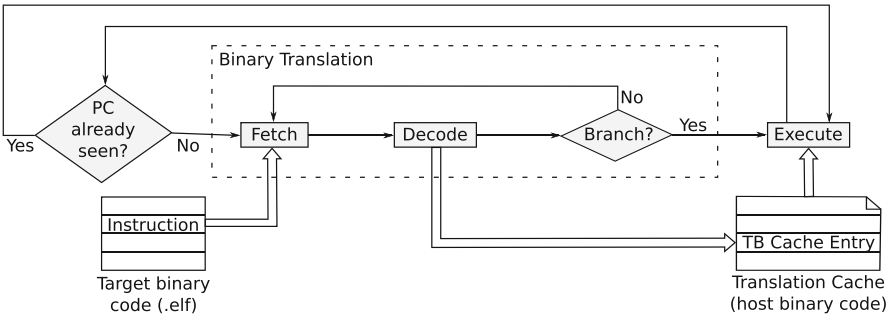


Fig. 18.4 Binary translation principle

The instruction corresponding to the program counter of the simulated processor is fetched from the target binary code. The fetched instruction is then translated into several host instructions. If the current instruction is not a branch instruction, the next target instruction is fetched and decoded. Otherwise, the binary translation stage ends. The sequence of instructions treated by the binary translation stage at once forms a Translation Block (TB). The host instructions generated for a translation block are grouped together and stored into the translation cache and are ready to be executed to simulate the corresponding target instructions behavior. Once executed, the simulator has stepped forward by one TB, and the simulated `pc` has evolved accordingly. The simulator then verifies the existence of the translation corresponding to this new program counter value. If it already exists in the translation cache, it is directly executed. The idea is that the price paid for host code generation will be amortized as the translated code sequences are usually executed many times.

The notion of translation block is similar in spirit to the notion of basic block used by compilers, but they are not identical. Even though translation blocks and basic blocks end after a branch instruction, there are other conditions that end only the translation blocks or only the basic blocks. As an example, a translation block is also ended at the page boundary of the target processor, because the DBT engine must ensure that the page is mapped in memory. Other conditions for ending a translation block include instructions generating exceptions (e.g., undefined instruction), change of the execution mode of the target processor, etc. By definition, the basic blocks are also ended before the instructions which are the target of jump or branch instructions.

A binary translation simulator would generate in this case a new translation block starting at the jumping address. So, an instruction can be part of several translation blocks, but only of a single basic block.

Given the simulation context we are interested in, it is required to be able to support different types of target processor architectures on different types of host architecture, even though simulation will very likely take place on a x86_64 machine. Given t target processor types and h host processor types, the approach just described leads to the development of $t \times h$ translators to allow all target processors to run on all the host processors.

Due to the complexity of writing a translator, the principle of retargetable DBT has been proposed [28, 29]. Instead of being translated directly to host code, each target instructions is first translated to a bytecode (also often called intermediate representation or IR) common to all targets. The virtual instructions of the bytecode, that we call *microoperations*, are then translated to host code. This way, target and host translations are independent. Adding a new target processor requires "only" a translator of the instruction set of that processor to the bytecode, no matter the number of hosts on which the new target processor can be simulated. By adding a translator from the bytecode to a new host processor, all target processors can be simulated on the new host processor. So, the number of translators is now $t + h$. The principle of these simulators, given Fig. 18.5, is close to that presented in Fig. 18.4. The target instruction decoder generates the bytecode corresponding to

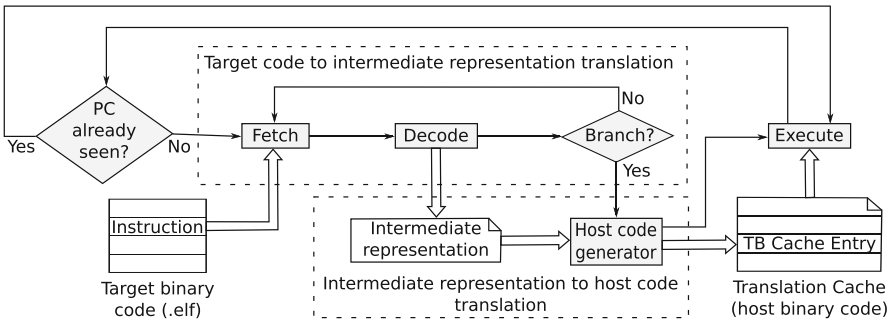


Fig. 18.5 Retargetable binary translation principle

Target code (MIPS)	Description	Generated micro-operations (QEMU)	Generated host code (x86_64)

`sltiu v1,v0,23`	v1 ← v0<23 ? 1 : 0	`mov_i32 tmp0,v0`
		`movi_i32 tmp1,$0x17`
		`setcond_i32 v1,tmp0,tmp1,ltu`
			`mov %ebx,0x74(%r14)`
			`cmp $0x17,%ebp`
			`setb %bpl`
			`movzbl %bpl,%ebp`
			`mov %ebp,0xc(%r14)`

Fig. 18.6 Microoperations and host code generated while translating a target instruction

the translation block under consideration. Then, the *host code generator* produces the host code corresponding to that translation block for the host processor.

Figure 18.6 is a simplified illustration of the translation process on a MIPS instruction, the intermediate bytecode is a 3-address Instruction-Set Architecture (ISA) close to the one of QEMU [12], and the target is x86_64. The generated bytecode uses a mix of processor architectural registers, e.g. `v1`, and of temporaries, here `tmp0` and `tmp1`. The architectural registers represent a part of the processor state and belong to a larger structure called the *environment* in QEMU. Their value survives between translation blocks, while temporaries live only inside a TB. Once translated as host code, these elements become offsets within the environment, pointed to by the host register `%r14` in QEMU for x86_64.

When performing translation, some instructions require complex operations that can hardly be produced by generating host instructions at run time. For example, when doing a load instruction, it is necessary to access the Memory Management Unit (MMU) model to determine whether the addressed page is mapped in memory. If not, a page fault exception must be raised. This leads to complex operations, like traversing the page table, which are done using a lot of code. Therefore, instead of generating the whole code dynamically (which is hardly possible and an overkill), the DBT engines generate calls to specific functions (called helpers) to handle these instructions.

Even though instruction interpretation is the core of dynamic binary translation, quite a lot of housekeeping is necessary to actually obtain a running simulator.

- First and foremost, as the guest OS expects a memory management unit, it is necessary to simulate it. It is also necessary to produce the expected page faults including the protection flags, etc. Page boundaries are handled at translation time; flags are handled on memory accesses.
- Second, self-modifying code must be supported. This is especially true as now, even in embedded environments, many just-in-time compilers are used (just think of Javascript in a browser or of Android). The performance issue related to supporting dynamic code generation in DBT is considered as of primary importance [17].
- Third, the translation cache has to be managed. A cache, by nature, has a finite size. The placement of the newly generated translation blocks and the replacement policy in case of overflow have to be determined.

- Fourth, multiprocessor systems must be supported. A basic implementation simply simulates the processors one after other in a predefined order by calling their execution function. The simulation of a processor is suspended, and the simulation of the next one (e.g. round-robin algorithm) resumed when an interrupt or an exception occurs. However these events can only occur at the border of a translation block. Other solutions can be thought of to better mimic parallelism, as explained in Sect. 18.5.
- Finally, many optimizations are possible. Chaining is a classical optimization which links a block to its successor using a jump instruction without going back to the DBT engine when possible. The identification of the mostly used paths, called hot paths, and the optimized retranslation of these paths can be beneficial. It can also be counterproductive, as it requires accounting in the translation blocks and time for optimized retranslations. Finding the right balance is known to be hard [12].

18.3 Support for Non-scalar Architectures

Dynamic binary translation is usually used for running target code for a scalar architecture on a scalar host. However, MPSoCs target specific markets with tight power and area constraints and therefore embed specialized processor extensions or processors with unusual architectures. The goal of this section is to briefly present how such features can be efficiently supported by DBT.

18.3.1 Support for SIMD Instructions

Single Instruction, Multiple Data (SIMD) instructions perform parallel operations on multiple data (Single Instruction, Multiple Data (SIMD)). There are today multiple ISA extensions providing SIMD instructions to general purpose CPUs. For performing parallel operations on multiple data, an SIMD instruction performs the operation (or sequence of operations) on registers interpreted as array of values. This array of data can have a variable number of values of various size, for example, a 128 bits wide register can be viewed as two 64 bits, four 32 bits, eight 16 bits, or sixteen 8 bits values. On top of that, the variety of the operations applied to the data is huge. It ranges from the classical arithmetic operation (add, sub, shift, ...) to saturated or rounding arithmetic. Among this large range of instructions, each SIMD instruction set represents a unique subset choice made by the processor architects. SIMD extensions can also integrate some exotic instructions such as polynomial multiplication or pixel distance computation that have no equivalent in other SIMD ISA.

DBT engines focus on efficiency for the most often used instructions, following the adage "make the common case fast and the uncommon case correct." As the SIMD extensions are rarely relevant in general purpose computing, most translators use helpers to execute their behavior even though all host processors include SIMD instructions.

People in companies developing processors [20] have looked at this issue as the number of different SIMD extensions for different processor generations (e.g., MMX, 3DNow!, SSE1, ..., AVX for Intel) is huge, so having legacy code being able to benefit from the most efficient version available on the actually running processor has a clear value for some applications [26].

Replacing the helpers by dynamically generated code which uses the host SIMD instructions opens the interesting question of the appropriate intermediate bytecode for representing SIMD instructions [21]. The main two constraints to which one may think for this bytecode are:

- limiting the number of new IR microoperations in order to limit the burden on the code generator and the overall performances of the binary translator,
- adding enough microoperations in the IR to allow a wide coverage of both target and host SIMD instruction sets.

Indeed, adding too many microoperations will tend to the addition of one microoperation per SIMD instruction. This will not solve the problem since the code generator (the second phase of DBT) will have a heavy work to do to produce the inlined optimized code for each of the SIMD microoperations. Conversely, if not enough microoperations are added, the semantic of all SIMD instructions will not be expressed, and the translator will have to perform this translation using many non-SIMD microinstructions. A simple way to extend the IR is to choose a set of microoperations which is close to the intersections of the more widely available SIMD instruction sets. The IR microoperations will be 3-address operations since it is the most general case and allows to represent the 2-address versions easily, whereas the reverse is not true.

As opposed to scalar DBT, finding instruction equivalence in SIMD DBT has to take care of the SIMD specificities: parallelism and register interpretation. The following section illustrates these peculiarities with concrete examples of translation from ARM NEON instruction set to Intel MMX/SSE.

Direct mapping between instructions: in the presence of an exact equivalence between a target SIMD instruction and an host SIMD instruction, the behavior of the SIMD DBT is quite similar to the one of the scalar DBT. This case can be called a *direct mapping*. The main difference between scalar and SIMD *direct mappings* lies in the fact that it is necessary to guaranty that there is the same level of parallelism between the two instructions, i.e. the same interpretation of registers (couple of number and size of the values).

This case is widely applicable on arithmetic operations of SIMD instruction sets. Figure 18.7 illustrates the DBT of an ARM Neon `vadd.i16` into an Intel MMX/SSE `paddw`. The IR microoperation used to propagate the parallelism is named `simd_128_add_i16` and represents the SIMD instruction performing 8 parallel adds on 16-bit values in 128-bit registers.

Table 18.1 gives some examples of *direct mappings* between the ARM Neon add instructions and the Intel SSE ones. These examples are only 128-bit adds, but equivalent mapping can also be found for 64-bit instructions and for other arithmetic instructions such as `sub`, `and`, `or`, and `xor`.

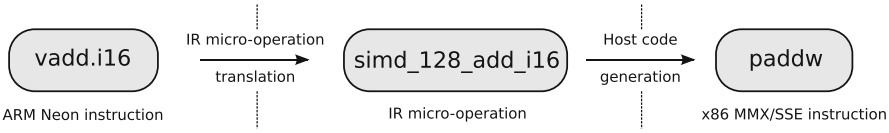


Fig. 18.7 Direct mapping between `vadd.i16` Neon instruction and `paddw` MMX/SSE instruction

Table 18.1 Mapping between addition instructions

Operation	Neon instruction	MMX/SSE instruction
Add 8 bits	`vadd.i8 Qd, Qn, Qm`	`paddb xmm1, xmm2`
Add 16 bits	`vadd.i16 Qd, Qn, Qm`	`paddw xmm1, xmm2`
Add 32 bits	`vadd.i32 Qd, Qn, Qm`	`paddd xmm1, xmm2`
Add 64 bits	`vadd.i64 Qd, Qn, Qm`	`paddq xmm1, xmm2`

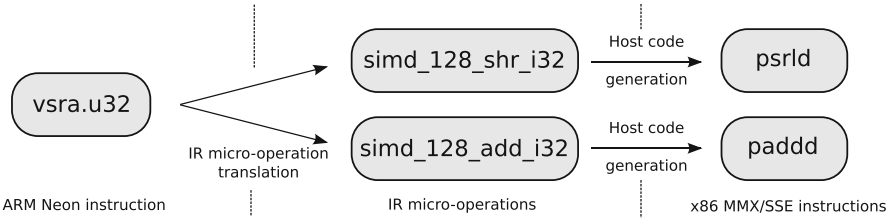


Fig. 18.8 The `vsra` Neon instruction is translated into two IR microoperations

No direct mapping: in a less favorable case, there exists no direct mapping between instructions of the instruction sets. Most of the cases, this lack of mapping is due to a lack of generality of the operations performed by the target SIMD instruction. In that case it is only of little interest to have a microoperation in the IR for that instruction. The strategy in such cases is to split the target SIMD instruction in more elementary operations already present in the IR. This technique is once more identical to the one used in scalar DBT, but more parameters have to be taken into account during the process, i.e. parallelism level and registers interpretation.

Figure 18.8 gives an example of this situation with the translation of the ARM Neon `vsra.u32` instruction which performs a right shift on operands and accumulate the shifted results in the output register. This SIMD instruction is translated into two elementary IR microoperations `simd_128_shr_i32` and `simd_128_add_i32`. The code generator can then find an equivalent for each microoperation, i.e. `psrld` and `paddd`.

Exceptional case: a third and least favorable case is finally possible. This situation occurs quite rarely, but due to the way instructions have been chosen for integration in the SIMD instruction sets, it can be encountered. It happens when an SIMD instruction of the target can be translated into a corresponding IR microoperation, but no equivalent is available in the host SIMD instruction set. As shown Table 18.2, all versions of the shift are available in ARM Neon

Table 18.2 Mapping between left shift instructions

Operation	Neon instruction	MMX/SSE instruction
shl 8 bits	`vshl.i8 Qd, Qm, #imm`	N/A
shl 16 bits	`vshl.i16 Qd, Qm, #imm`	`psllw xmm1, xmm2`
shl 32 bits	`vshl.i32 Qd, Qm, #imm`	`pslld xmm1, xmm2`
shl 64 bits	`vshl.i64 Qd, Qm, #imm`	`psllq xmm1, xmm2`

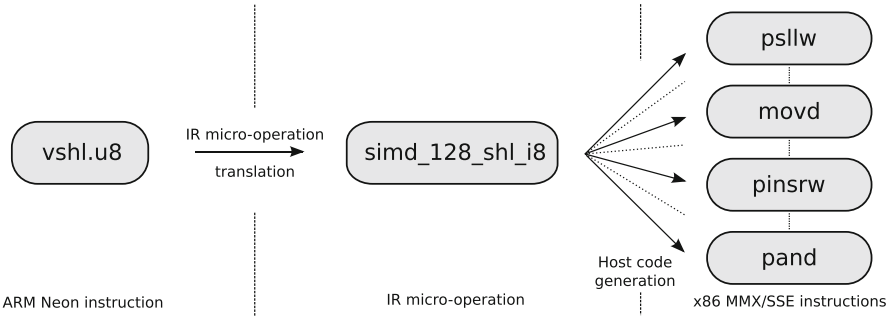


Fig. 18.9 The left shift microoperation is translated into multiple MMX/SSE instructions

SIMD instruction set. This is even true for all SIMD instruction sets that have been analyzed for this study, except for the Intel SSE SIMD instruction set. As it can be realized from this table, there exists no instruction for shifting 8-bit values. As this operation is present in all other instruction sets, it is present in the IR.

The code generator has then to solve this situation by generating multiple host instructions, as shown Fig. 18.9. The example given in this figure is for the translation of a 8-bit logical left shift emulated by a 16-bit version.

Comparison instructions: as far as comparisons are concerned, PowerPC Altivec, Sparc VIS, MMX/SSE, and Neon instruction sets provide the result for each element in the output operand, whereas the MIPS DSPASE sets flags. Because of this unbalanced distribution, a reasonable choice is to define microoperations producing their results in the output operand.

18.3.2 Support for VLIW Architectures

VLIW are not uncommon in the embedded space; indeed several recent many-core architectures are VLIW based [10, 14, 18], as they provide a high computing vs. power efficiency. The VLIW idea is to make a processor simple yet powerful by having the compiler provide the Instruction-Level Parallelism (ILP) explicitly in the execute packet, i.e., the set of instructions to be executed concurrently. The VLIW implementation choices are mainly trade-offs regarding simplicity of the design against compiler complexity, the major one being bypasses and stalls against register update latencies (also called delay slots).

18.3.2.1 VLIW Specificities

VLIW have specificities which render the VLIW to scalar DBT process particularly hard. The main characteristic of the execute packet is that multiple member instructions are meant to be executed in parallel. Some may use a register as input operand, while another one might use it as an output operand. The correct behavior is to feed the input operands with values the register had before the execute packet begin, which is not trivial in the DBT context.

The next characteristic concerns the latency of arithmetic and logic instructions, which may be greater than one cycle. Instead of stalling the pipeline until the availability of the result in the destination register, the compiler has the responsibility to ensure that an instruction depending on this result will not be scheduled before the end of the delay. It also means that all instructions executed in between can read this register to get its old value, and it is even possible to write it, as the actions will occur after the corresponding latency, as long as there is no multiple writers on the same destination register at the same cycle (otherwise, the result is undefined).

The last specificity is related to branches. Branch instructions have also delay slots, which means that instructions following a branch will be executed irrespective of the branch outcome.

18.3.2.2 VLIW DBT Extension Principles

The DBT has to handle the fact that multiple instructions are executed in parallel which may have destination registers being source registers of others in the same execute packet. The Write-After-Read (WAR) dependencies must be fulfilled to obtain correct behavior against execute packet semantic. The systematic solution to this problem is to introduce a new copy of a register each time it is overwritten. This strategy is similar to the Static Single Assignment (SSA) [9] form used in compilation when considering target registers as variables.

Applied to the intra-execute packet WAR dependency, this leads to at most two living versions of the register in each execute packet, one corresponding to the old value and one corresponding to the updated value.

Although this solution clearly solves the WAR problem, it does not solve the issue of instructions delay slots. The solution is again relatively simple: the old version of a register must be kept and used in case of a read until all delay slots of the instruction have been consumed. This results in keeping possibly alive more than two versions of the same register at the same moment, which is clearly not the case in conventional SSA.

The number of living versions of the registers increases, but fortunately, this number is bounded to a reasonable amount, which is the maximum instruction latency among all instructions plus 1. Indeed, the worst case is a sequence of largest latency instructions writing to the same register. In that case, one living version must exist for each instruction executed between the execution of the first occurrence and the availability of its result, plus one version for the previous value.

Finally, the branches have to be handled in two phases. Firstly, the code responsible for the computation of the branch target address is generated when

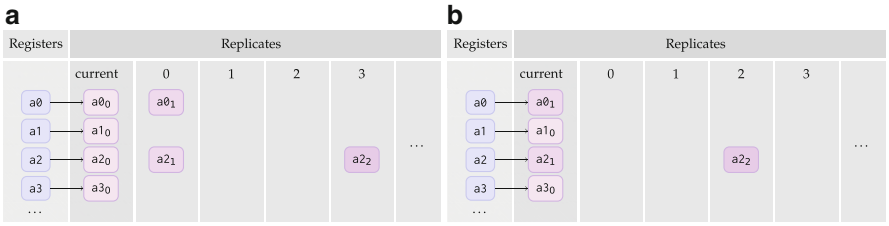


Fig. 18.10 Reading and writing registers replicates

encountering the branch instruction. Secondly, a TB ending instruction is produced after the delayed instructions and `pc` is updated accordingly.

18.3.2.3 TB Entry and Exit States

Using the above-described techniques (plus others not described here but similar in spirit, to handle, e.g., predicated instructions), the DBT translation phase can produce a sequential bytecode using registers replicates accomplishing the exact functional behavior of the source VLIW code.

For implementation purposes, each register points to the head of a queue. Reading the register value is done accessing the head, while the registers replicates (created when an instruction writes the register) are inserted in the position representing their latency. Figure 18.10 illustrates the idea. At first, replicates 0 are created in case a read occurs. Then, depending on the execute packet instructions, two 0 cycle latency instructions writing registers 0 and 2 and a 3 cycle latency instruction writing register 2 Fig. 18.10a, the replicates are created. After the translation of an execute packet, the queues are shifted left one position (see Fig. 18.10b), the leftmost replicate being kept if no value overwrites it, and the useless replicates are freed.

Due to this translation time renaming strategy, the working version (currently used versions) of the registers, although known, is unpredictable. More precisely, as TB are translated independently, the working versions when leaving a TB are unknown to the newly entered TB. Indeed, when a TB has several different predecessors, there is no way to guaranty that the working version will be identical at the exit of all the preceding TBs, and the translator does not even know if a TB has more than one predecessor when performing the translation.

The solution for handling this need of TB independence is to define a canonical entry and exit working register set for TBs. In that way, once translated, the TB will be reusable from all TBs pointing to it. The canonical state will be composed of the first replicate of each register, in which are mapped the first version of each register at the beginning of TB translation.

A further, this time dynamic, complication may arrive: in some cases, the delay slots cross the border of a TB. In that case, an external mechanism needs to be set up to handle these delay slots. This mechanism needs to be external to the translator because of the TB independence requirements. Indeed, the translator loses

the information about delay slots when exiting a TB, and thus only a run-time mechanism can manage the delay slots in that case.

This mechanism records the pending delayed registers with their current cycle delay when exiting the current TB at run time. The next TB executed consults the recorded data during its first execution cycles (bounded by the maximum possible latency) to check if there are some registers needing to be updated as they reach their latency. This can be done through helper calls inserted by the translator.

18.3.2.4 Complexity of the Modifications

From a pure functional point of view, all modifications needed to implement the VLIW DBT are part of the translator. The code generator does not strictly need to be modified to handle these new features.

The modifications needed are first a change from a simple array of registers in which each cell represents a target register to an array of these registers in which a line represents all the replicates of one target register. This change requires to modify all mechanisms at translation time to allocate the correct replicate each time an assignment to a target register occurs.

The delay slot handling, delaying use of a newer replicate, can be modeled using a simple queue, in which future versions of registers can be inserted at the corresponding delay. At the end of each cycle, all delayed register versions progress of one cycle in the delay queue. All these computations occur at translation time. Complexity is once again limited.

The handling of the canonical state implies a modification of the translator but impacts the generated code. Indeed, the easiest way to return to the canonical state at the end of a translation block is to generate instructions that move current working replicates of registers to the ones defined in the canonical state. This solution has a limited complexity on the translator but has the unfortunate side effect of increasing the TB size and thus its execution time.

Finally, the handling of delay slots crossing TB edges is a pure run-time mechanism. This mechanism has to be identical for all TB to be valid for all sequences of TB. The amount of generated code for this purpose is not huge, as it consists of generating helper calls to propagate correctly the register updates. Even though the functions themselves are quite straightforward since they only handle registers updates through a run-time delay queue similar to the one described before, the run-time overheads necessary to set up and perform these calls are quite large compared to a simple sequence of generated host instructions.

18.4 Annotations in Dynamic Binary Translation

Although DBT is a very efficient technique for instruction interpretation, it is not, in its usual form, suited to performance evaluation of software, making the virtual platforms built on top of it unsuited to design space exploration of hardware/software systems. Indeed, the translation process produces host code

Target code (MIPS)	Generated micro-operations (QEMU, simplified)	Generated host code (x86_64, simplified)

`sltiu v1,v0,23`	`ld_i32 tmp0,env,$0x48c`
	`add_i32 tmp0,tmp0,1`
	`st_i32 tmp0,env,$0x48c`
	`mov_i32 tmp1,v0`
	`movi_i32 tmp2,$0x17`
	`setcond_i32 v1,tmp1,tmp2,ltu`
		`mov 0x48c(%r14),%eax`
		`add $1,%eax`
		`mov %eax,0x48c(%r14)`
		`mov %ebx,0x74(%r14)`
		`cmp $0x17,%ebp`
		`setb %bpl`
		`movzbl %bpl,%ebp`
		`mov %ebp,0xc(%r14)`

Fig. 18.11 Generation of annotated code

whose behavior must reproduce the one of the target code, and there is no provision to estimate any kind of information (time, power, etc.) related to the execution of a translated block. However, as translation generates code, it is possible to produce nonfunctional code which aim at doing, during execution, some specific activity [6,30]. These nonfunctional pieces of code are called *annotations*. The first goal of these annotations is to make the simulated processors accurate from the point of view of the time internally required for instructions execution.

The place at which these annotations are exactly placed depends on the overall virtual prototyping approach, but it can be either before the first functional instruction of a translation block, before or after a specific instruction, or after the last functional instruction of a translation block.

Assuming the annotation targets a rough estimation of the software execution time (excluding cache and memory effects), a first approach consist of generating, before the translation of the instruction, an addition on a global variable, as illustrated Fig. 18.11, which can be compared with Fig. 18.6 to measure the overhead due to annotation.

In this case, the number of cycles is a field at offset `0x48c` in the environment.

The value of the field is incremented by the number of cycles associated to the instruction, quantity typically found using a look-up table that contains the information copied from the processor datasheet. Sometimes the number of cycles required by an instruction depends on values available only when the instruction is executed. For instance, the number of cycles required by a multiplication instruction may depend on the values of the operands, which may differ from an execution to another. In that case, specific code has to be generated to add the corresponding difference to the preceding value. Since entering a translation block guarantees that all instructions it contains will be executed, a wiser way, that does more at translation time but less at execution time, consists of generating code at the end of the translation block that adds to a variable the value accumulated during the translation of the whole translation block.

Annotations can also be used for power estimation, or for totally different purposes, such as fault injection for code analysis [3] or generation of traces for analysis [8].

18.4.1 Cache Modeling Strategies

A modification which greatly improves the time accuracy (at the cost of execution speed) consists of modeling the instruction and data caches. This section deals with level-1 (L1) caches, as the situation with upper-level caches depends on the structure of the memory hierarchy. Indeed, if the level-2 (L2) caches are private, then the same approach as for L1 cache, described below, should be used. However, if they are shared, then a global state must be visible, so that correct updates take place. The simulation of the L2 cache can either be a component at TLM level or a shared structure within the DBT engine. In the former case, it takes benefit from the event-driven nature of the TLM simulator to synchronize with the rest of the system as any other component would do, at the cost of more synchronizations within the simulator. In the latter case, the relative progression of all hardware models must be considered carefully.

For the instruction cache, the access to the models occurs thanks to a helper called through a microoperation inserted at the beginning of each translated translation block and, inside a translation block, before the first instruction from each instruction cache block. As exact target instruction addresses are known at translation time (even for dynamically generated code since it is translated at run time), this can always be done at relatively low cost.

Data caches using write-through or write-back policies can also be modeled for main memory read/write data accesses. Each time a main memory location is read or written, its presence in the data cache is checked and the proper action (return value, fetch block, write-allocate or write update) taken. Here also the addresses are exact, so the modeling in terms of hits and misses by a simple array of addresses can be faithful.

For set associative caches, the replacement policy may be difficult to reproduce exactly, e.g., when a set is chosen at random with a generation of the random number depending on a free-running counter, but this is not DBT specific.

Figure 18.12 shows how the annotations are inserted to handle the computation of cache misses. To limit the code size, pseudo-code instead of actual code is used. In this figure, the first column is the instruction address, the second column the generated code without annotations, and the third column the generated code with annotations. Assuming a 4 32-bit words long cache block, it is necessary to check the presence of the instruction in the cache only when the 4 lower bits of the address are equal to zero, which is done by a call to the `insn_cache_verify()` helper when the program counter has this property. Read and write accesses, even though handled specifically by the translator back-end to simulate the MMU and actually access the memory and peripherals, are simple microoperations in the front-end. To model the L1 data cache, helper calls (`read_access()` and `write_access()`) are added.

When modeling shared memory processor subsystems, a further issue is cache coherency. Two solutions can be thought of for maintaining coherent data or

Insn addr	Target code (pseudo-code)	Generated code (pseudo-code)	Generated annotated code (pseudo-code)

10	insn1_reg_operation	host_insn1_for_insn1
		.....
		host_insnN1_for_insn1
			insn_cache_verify(0x18);
			nb_cycles += cpu_datasheet[insn1];
			host_insn1_for_insn1
			.....
			host_insnN1_for_insn1

14	insn2_load_from_addr1	host_insn1_for_insn2
		.....
		host_insnN2_for_insn2
			nb_cycles += cpu_datasheet[insn2];
			read_access(addr1);
			host_insn1_for_insn2
			.....
			host_insnN2_for_insn2

18	insn3_reg_operation	host_insn1_for_insn3	nb_cycles += cpu_datasheet[insn3]; .....
1c	insn4_reg_operation	host_insn1_for_insn4	nb_cycles += cpu_datasheet[insn4]; .....

20	insn5_store_x_to_addr2	host_insn1_for_insn5
		.....
		host_insnN3_for_insn5
			instr_cache_verify(0x20);
			nb_cycles += cpu_datasheet[insn5];
			write_access(addr2, x);
			host_insn1_for_insn5
			.....
			host_insnN3_for_insn5

Fig. 18.12 Cache modeling through annotation

instruction (necessary to support dynamically generated code) within the caches. The brute force one simply ignores the cache protocol and traverses all cache models when a write occur to check for the presence or absence of the written address in the cache. When the number of processor is small, this traversal is quick enough to be acceptable. To do so, the run time of the DBT engine simply needs to maintain a list of caches accessible by all CPUs. However, when the number of processor increases, the traversal time becomes unacceptable, and the solution is then to implement (even in a simplified form) a cache coherence protocol. Indeed, assuming n is the number of writes and p the number of processors, traversing all caches is in $O(p \times n)$, while a hardware cache coherence protocol would lead to performance in $O(k \times p)$, with $k \ll 10$ in average since the number of sharers of a data is usually low. This behavior can be mimicked efficiently by accessing a hash table indexed by the hashed address whose entries contain the address as key and a pointer to the array representing the cache which caches the address as data. The tipping point between both solutions depends on the implementation details, but it seems clear that for many-core architectures, the latter one is more efficient.

As can be seen, adding cache models increases the size of the generated code and requires more complex handling of the memory accesses at execution time, which leads in any case to an important degradation in simulation speed. Given the fact that is it possible to count load and store, higher, typically analytical, models can also be used when accuracy can be trade-off for speed.

18.4.2 Modeling Branch Predictors

Even though not that many embedded processors today include branch predictors, it is worth taking a look at how it can be modeled in DBT-based simulation as the influence of this microarchitectural feature on out-of-order execution performance is major. At first, modeling a branch predictor seems easy: the branch outcomes are known, so updating a set of branch history tables (as in the TAGE [27] predictor or its descendants, current state-of-the-art predictors) to predict a future branch outcome is straightforward. However, branch predictors use large global tables, shared by all branches in the execution flow. As opposed to caches, these tables are accessed one after the other at what looks like random indexes (computed as hashes of branch history and branch instruction address). As far as simulation is concerned, repeated accesses to random places lead to poor program locality and, in consequence, poor simulation performance. Experiments have shown slowdowns of 1.5× to 15× as compared to raw DBT execution [13]. To limit this overhead, a performance/accuracy trade-off can be made by defining models which predict the behavior of the branch predictor. The principle consists of transforming the tables of the reference architecture, which are global, i.e., shared between all branches, into information local to each branch. This leads to allocating the table entries only when needed and enhances locality, thus limiting the number of cache misses when simulating the predictor model.

Taking TAGE as an example, the bimodal table, as can be seen Fig. 18.13, is reduced to a 2 bit counter which can be stored in the local data of each branch. Then, for the tagged tables, two parts are distinguished, the tables themselves and the way they are indexed. Concerning the tables, a simple solution is to use a single tagged table, ignoring history length. In the same way as for the bimodal table, only

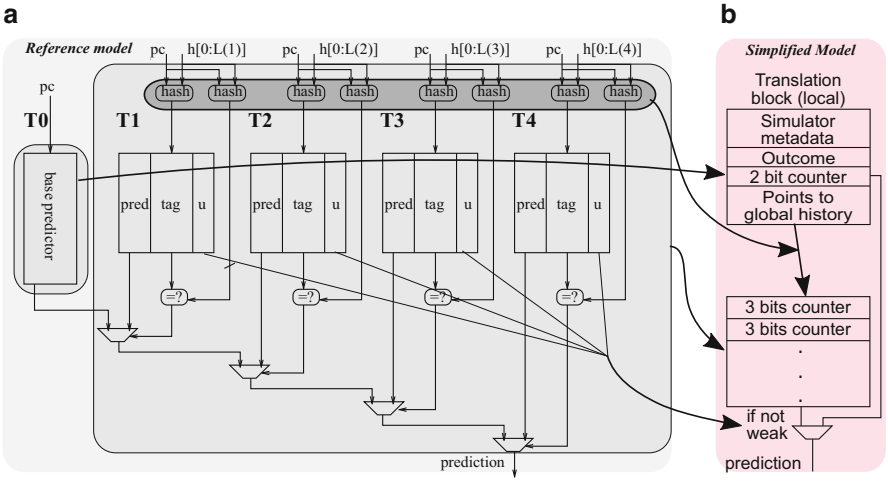


Fig. 18.13 (**a**) Reference [27] and (**b**) simplified models for a TAGE branch predictor

the potentially reachable entries, i.e., the ones concerning already seen branches, can be allocated locally for each branch. This results in a local table that is indexed by the global history, which, in the reference architecture, was hashed with the program counter (PC) of the branch.

The choice of the entry is then naturally simplified as there is only one, local, tagged table to choose from.

The confidence state of the entry in the tagged table is used to select between the tagged table and the bimodal table. If it is "weak," the prediction will be given by the bimodal table; otherwise it will be given by the tagged one. This should ensure that, as in the reference architecture, tagged prediction is chosen only when the tagged entry is "sure" of its prediction. The tag of the tagged table is not useful anymore, as entries concerning one branch are now specific to it. In other words, aliasing, i.e., multiple branches pointing to the same entry, already very unlikely thanks to tags, is impossible in the simplified model.

The resulting model uses memory local to the current basic block, which tends to enhance locality, but it also uses more memory, as information shared due to aliasing is now duplicated. The information used by the simplified predictor is similar to that of the reference architecture: the outcome of the current branch, its own data, a global history of branch outcomes, and, finally, the structure of the already executed code (to store data per branch). This model produces identical predictions to the TAGE architecture in average 95% of the time for a 5% execution time overhead. Even though 95% identical predictions may seem a good achievement, the impact on the execution time may be of importance, so finer models can also be of interest.

As for caches, a wide range of models can be used for taking into account branch prediction, and it boils down to produce annotations at the end of the basic blocks, as shown Fig. 18.14. The `bp_model` helper is called with the address of the branch instruction (`PC`) and the outcome of the branch (`bcond`).

18.5 Integration with TLM Simulations

Integrating DBT-based Instruction-Set Simulator (ISS) in a TLM simulation environment can be done in two steps, as depicted Fig. 18.15. First, all processors belonging to a Symmetric Multi-Processing (SMP) subsystem are grouped together

Target code (MIPS)	Generated micro-operations (QEMU, simplified)
`...`	`mov_i32 tmp0,t3`
`beqz t3,0x8000024c`	`movi_i32 tmp1,$0x0`
`nop`	`setcond_i32 bcond,tmp0,tmp1,eq`
	`movi_i64 tmp2,$bp_model`
	`call tmp2, PC, bcond`
	`movi_i32 tmp1,$0x0`
	`brcond_i32 bcond,tmp1,ne,$0x1`
	`...`

Fig. 18.14 Annotation to model branch prediction

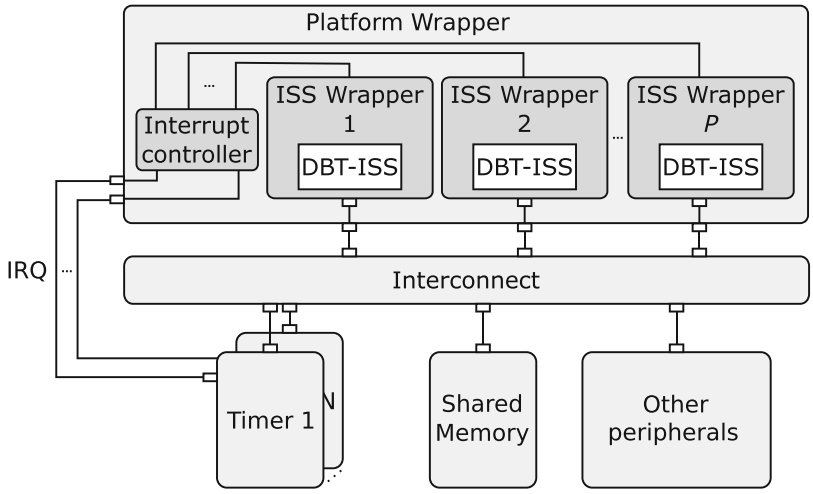


Fig. 18.15 Binary translation based simulator – SystemC simulation platform

in a module that can be instantiated by the TLM simulator (*platform_wrapper*). It is assumed that all processors are identical and have identical cache geometries and that they have a shared view of the memory. This allows to share the translation cache between them, avoiding retranslation on one processor of code migrating from another processor in SMP systems and an efficient implementation of cache coherency.

Second, the platform wrapper instantiates a module wrapper for each processor (*iss_wrapper*). The execution of each processor is performed in the context of the process of its wrapper. This way, the processors are simulated concurrently by the TLM simulator time-sharing scheduler. The platform wrapper is useful for managing the common aspects shared by the processors (e.g., inter-processor interrupt management, platform specific registers, interrupt controllers etc.).

The platform wrapper is connected to an interconnect, through which it can communicate with other hardware components (memory, timers, DMA engines, frame buffer, etc.) also connected to it. All hardware components are implemented as TLM modules.

From the initial DBT platform, the platform uses only the processor models with, if required, their MMUs. All other devices are externalized and implemented as TLM modules. This allows to enforce the notion of IP and reuse, thanks to the TLM principles, whereas devices in DBT are described in very ad hoc manner and use shortcuts hardly acceptable when the simulation is also used for design space exploration purposes. The main memory is also implemented as one or more TLM modules. For accessing TLM models other than the main memory, a few ranges of the simulated processor physical addresses are mapped as I/O addresses in the processor wrapper. The I/O requests from the simulated processors are then transformed by the processor wrappers into TLM requests, using the protocol understood by the interconnect. Memory, on the other hand, is accessed through

a direct memory interface. This avoids relinquishing the CPU to the TLM part of the simulation and thus saves a lot of time.

18.5.1 Precision Levels

Depending upon the accuracy one expects from the simulation, four trade-offs can be made regarding memory accesses.

The first approach does not implement caches and uses the main memory internally allocated by the DBT engine. The time required for executing the number of cycles corresponding to the instructions simulated is consumed using the *wait* function of the simulator. In this configuration, it is considered that the memory is always available for all processors, without any cycle cost for accessing it. The communication with other peripherals is performed by sending requests over the interconnect. The time consumed for these accesses is composed of the time consumed by each TLM components involved in the transmission and the reception of the request packets. In this case, a simulated processor synchronizes with the rest of the TLM platform only when an I/O operation is executed and when that processor is unscheduled by the DBT simulator. Due to the reduced number of synchronizations, large pieces of translated code are executed without interruption. As a result, the simulation will be very fast (close to the DBT alone). The accuracy in this case will be low as the cache effects and the time required to communicate with the main memory over the interconnect are not accounted for. Since all memory accesses are done without going through the interconnect, there is no need for an explicit support for cache coherent mechanisms.

The second approach relies on the caches being implemented only from the hit/miss point of view, while the main memory of the initial DBT engine is still used. As opposed to native simulation or compiled simulation presented in ▸ Chap. 19, "Host-Compiled Simulation", dynamic binary translation uses the exact addresses for instructions fetch and data accesses. However, the target instructions are fetched from memory only once, for translation, before their first execution. The simulator always executes the generated binary host code stored in the translation cache. So, to accurately account for these accesses, a model of the cache is needed. Both data and instruction caches can be modeled as pure directories, so that an array access (with the proper tag, index, and offset) indicates if the instruction would in reality be in the cache. A cache miss issues a TLM *wait* for a time precomputed to be required to load a cache line, without actually sending the request over interconnect. As for the previous approach, the I/O operations involve the interconnect and other TLM hardware models. The time corresponding to the simulated cycles is consumed at the beginning of the next synchronization. In this case, the processors are synchronized with the TLM simulator when a cache miss occurs, an I/O is executed or when they are unscheduled by the DBT engine. The simulation speed for this configuration is reduced a lot because of the large number of synchronizations produced by the cache misses. As the precomputed time is consumed directly in the cache model, a single timed event is generated for each cache miss. This is not much, considering

that the transfer of a single byte over the interconnect requires more than 10 timed and untimed events. The accuracy increases by using a precomputed average value for the time required for a memory transfer over the interconnect. However, the interconnect load, hardly guessable as it is highly nonlinear when congested, is not taken into account.

The third approach is an extension of the second one, in which, instead of consuming the precomputed time when the cache misses occur, the consumption of time is postponed until the next synchronization produced by an I/O operation or by the normal unscheduling of a processor. At the synchronization moment, the sum of the precomputed times required by all write accesses and cache misses that have occurred since the previous synchronization is consumed. This way, the number of synchronizations is reduced, increasing simulation speed. The chances of preventing other processors to modify a variable waited on by the current simulated processor are higher in this configuration compared to the previous one because of the small number of synchronizations. This may have a negative impact on simulation speed as more cycles have to be simulated by a polling processor and has a negative impact on simulation accuracy.

The fourth and most accurate approach fully implements the caches and uses an external TLM memory module as main memory. In this case, in addition to the directory, the caches also have their data part. However, the data of the instruction cache is ignored. The instructions needed for translation are searched directly in the memory module, without issuing a TLM *wait*. The loading price will be paid by the instruction cache when the generated code is executed from the translation cache, as explained Fig. 18.12. In all cases, a cache miss issues a request over the interconnect. The simulation speed for this configuration will be even slower, because the requests and responses pass through all the components that are required for the transfer.

18.5.2 TLM Synchronization Points

In this DBT+TLM integration approach, a processor is simulated as long as it does not communicate with the world behind its caches and the DBT engine does not stop it. When an instruction/data cache misses or an I/O occurs, the processor simulation stops, and the processor wrapper synchronizes itself with the rest of the platform by consuming the estimated time required by the real processor to execute the instructions simulated since the last synchronization. In case no such event occurs, in order to limit the divergence between the different processors' execution, a synchronization can be forced after a predefined period of time without synchronization. For the target processor instructions designed for synchronization of the software running on a SMP architecture (e.g., exclusive load and store, compare, and swap), a synchronization should also be generated.

The processors' simulation order depends on the time consumed by the processors at synchronizations. A synchronization condition may occur at any time during the execution of a translation block (e.g., cache miss); thus unscheduling does not anymore always occur at the boundary of a translation block. As the DBT engine

unschedules the processors only at the translation block border, it is necessary to save their "execution context" before synchronization and restore it afterward.

18.5.2.1 TLM Synchronization After Long Intervals Lacking in Synchronization

Due to the fact that simulated processors do not synchronize at each memory access, if two or more simulated processors read/write from/to the same memory address, the instructions executed by these processors may differ from those executed on a cycle accurate platform. A write to an address should invalidate the corresponding cache line in all caches but the one of the writing processor, and these other processors should see the new value at their next read from that address. Because of the direct access to memory, the write is visible by the rest of processors before it happens in the simulated timeline, if the writing processor is simulated before the reading one. If the processors' simulation order is inverted, the reading processor does not see the effect of writing until it synchronizes and the writing processor executes the writing code.

Processor unscheduling after a predefined time without synchronization is needed even for cases when a processor waits in a loop for a simple variable to be changed by another processor or any initiator of the system or even by an interrupt handler on the same processor. This kind of loops would also prevent the interrupts to occur for that processor because the interrupt pending flag is set during synchronization. For example, for computing the processor speed, Linux waits in a loop for the *jiffies* variable to be incremented by the timer interrupt handler. The condition of unscheduling due to lack of synchronization is verified at the beginning of each translation block. The time period for this unscheduling condition determines the maximum lag of the interrupts.

18.5.2.2 TLM Synchronizations Caused by Target Synchronization Instructions

The threads of a software application usually synchronize together. A spin lock is an example of a software synchronization mechanism. The lock and unlock function of the spinlocks are usually implemented using exclusive load and store instructions.

Figure 18.16 presents an example of software running on two processors and using a spinlock for the software synchronization. This figure shows what would happen if the simulator would not generate a synchronization for this type of target instructions or if the spinlock functions would not be implemented using exclusive access instructions.

Figure 18.16a presents the execution on a real hardware. The first processor ($P1$) locks a spinlock at $t1$, at $t2$ a cache miss occurs, at $t4$ $P1$ releases the spinlock, and at $t6$ it executes an I/O operation. The second processor tries to lock the spinlock ($t3$) just before $t4$; it actually obtains the lock at $t4'$ and releases the spinlock at $t5$.

The execution on our platform in the case when the simulator would not generate synchronization for the exclusive accesses is depicted in Fig. 18.16b. Considering that $P1$ is first scheduled for simulation, it locks the spinlock at $t1$ without being unscheduled (the spinlock is placed in the main memory), but at $t2$ it is unscheduled

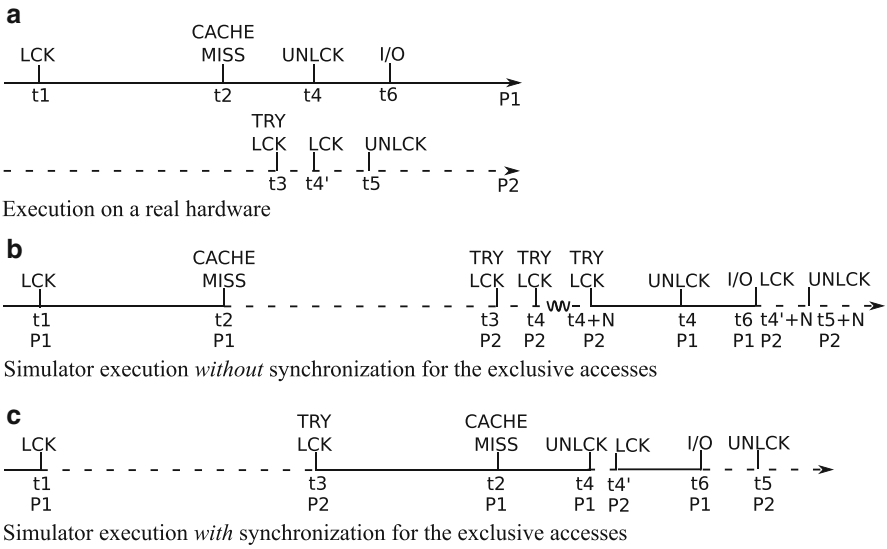


Fig. 18.16 Simulation behaviors based on the spinlock implementation

for synchronization before loading the cache line. $P2$ is now scheduled and at $t3$ it begins trying to take the lock. $P2$ reads in an infinite loop the spinlock locked value from the main memory.

After the predefined time without synchronization, $P2$ would be however unscheduled at time $t4 + N$. Then, $P1$ is scheduled, it unlocks the spinlock, and then it is unscheduled for synchronization before the I/O operation. $P1$ will be able now to take the spinlock, but it has oversimulated by time N.

The simulation behavior when spinlocks functions use exclusive access functions and the synchronizations that are generated for them are presented in Fig. 18.16c. In this case, the processors synchronize before each lock and unlock. $P1$ synchronize at $t1$ and $P2$ is scheduled. $P2$ is unscheduled before its first lock attempt. $P1$ synchronizes at $t2$, but it is rescheduled because $P1$ is more advanced in simulation time ($t3 > t2$). At $t4$, before releasing the lock, $P1$ synchronizes and it is unscheduled ($t4 > t3$). Between $t3$ and $t4$ (the simulation time of $P1$), $P2$ synchronizes and it is rescheduled at each attempt to lock the spinlock. After $t4$, $P1$ is scheduled and it releases the lock and it is simulated until $t6$. $P2$ gets the lock at $t4'$ (immediately after $P1$ has released it) and release it at $t5$.

18.6 Concluding Remarks

Dynamic binary translation provides a real increase in performance as compared to instruction accurate instruction-set simulators. This enhancement comes at the price of a much greater implementation complexity and, intrinsically, less capabilities to

monitor precisely nonfunctional properties of the software. The progress toward the integration of more and more cores on SoC makes it however a must for achieving hardware/software simulation at acceptable speed.

To act as an independent processor simulator, DBT must be integrated into standard event-driven simulation environments, e.g., SystemC. Then, it must support processors with non-scalar architectures and possibly more efficiently scalar processors [19], by resorting to run-time optimizations including dynamic recompilation. And finally, it should be capable of integrating models of processor specific microarchitectural details so that software performance evaluations (mainly timing and power) can be done.

References

1. Aarno D, Engblom J (2014) Software and system development using virtual platforms: full-system simulation with wind river simics. Morgan Kaufmann, Waltham
2. Baraz L, Devor T, Etzion O, Goldenberg S, Skaletsky A, Wang Y, Zemach Y (2003) Ia-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on itanium®-based systems. In: Proceedings of the 36th annual IEEE/ACM international symposium on microarchitecture, pp 191–201
3. Becker M, Baldin D, Kuznik C, Joy MM, Xie T, Mueller W (2012) Xemu: an efficient qemu based binary mutation testing framework for embedded software. In: Proceedings of the tenth ACM international conference on Embedded software. ACM, pp 33–42
4. Bell JR (1973) Threaded code. Commun ACM 16(6):370–372
5. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of the 1st IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis. ACM, pp 19–24
6. Cmelik B, Keppel D (1994) Shade: a fast instruction-set simulator for execution profiling. In: Proceedings of the 1994 ACM SIGMETRICS conference on measurement and modeling of computer systems, pp 128–137
7. Creasy RJ (1981) The origin of the VM/370 time-sharing system. IBM J Res Dev 25(5): 483–490
8. Cunha M, Fournel N, Pétrot F (2015) Collecting traces in dynamic binary translation based virtual prototyping platforms. In: Proceedings of the 2015 workshop on rapid simulation and performance evaluation: methods and tools. ACM, p 4
9. Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK (1991) Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst 13:451–490
10. de Dinechin BD, Ayrignac R, Beaucamps PE, Couvert P, Ganne B, de Massas PG, Jacquet F, Jones S, Chaisemartin NM, Riss F, Strudel T (2013) A clustered manycore processor architecture for embedded and accelerated applications. In: IEEE high performance extreme computing conference. IEEE, pp 1–6
11. Deutsch LP, Schiffman AM (1984) Efficient implementation of the smalltalk-80 system. In: 11th ACM SIGACT-SIGPLAN symposium on principles of programming languages, pp 297–302
12. Duesterwald E, Bala V (2000) Software profiling for hot path prediction: less is more. ACM SIGARCH Comput Archit News 28(5):202–211
13. Faravelon A, Fournel N, Pétrot F (2015) Fast and accurate branch predictor simulation. In: Proceedings of the design automation and test in Europe conference. ACM, pp 317–320
14. Flamand E (2009) Strategic directions towards multicore application specific computing. In: IEEE/ACM conference on design, automation & test in Europe, pp 1266–1266

15. Ghenassia F, Clouard A (2005) TLM: an overview and brief history. In: Ghenassia F (ed) Transaction level modeling with SystemC: TLM concepts and applications for embedded systems. Springer, Dordrecht
16. Gligor M, Fournel N, Pétrot F (2009) Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In: Proceedings of the 7th IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, Grenoble, pp 71–80
17. Hawkins B, Demsky B, Bruening D, Zhao Q (2015) Optimizing binary translation of dynamically generated code. In: Proceedings of the 13th annual IEEE/ACM international symposium on code generation and optimization. IEEE Computer Society, pp 68–78
18. Lethin R (2009) How vliw almost disappeared-and then proliferated. IEEE Solid-State Circuits Mag 1(3):15–23
19. Leupers R, Eeckhout L, Martin G, Schirrmeister F, Topham N, Chen X (2011) Virtual manycore platforms: moving towards 100+ processor cores. In: Design, automation & test in Europe conference & exhibition (DATE), 2011. IEEE, pp 1–6
20. Li J, Zhang Q, Xu S, Huang B (2006) Optimizing dynamic binary translation for simd instructions. In: Proceedings of the international symposium on code generation and optimization, pp 269–280
21. Michel L, Fournel N, Pétrot F (2011) Speeding-up simd instructions dynamic binary translation in embedded processor simulation. In: Proceedings of the design, automation & test in Europe conference, pp 277–280
22. Mitchell JG (1970) The design and construction of flexible and efficient interactive programming systems. PhD thesis, Carnegie-Mellon University, Pittsburgh
23. Monton M, Carrabina J, Burton M (2009) Mixed simulation kernels for high performance virtual platforms. In: Forum on specification & design languages, pp 1–6
24. Pétrot F, Fournel N, Gerin P, Gligor M, Hamayun MM, Shen H (2011) On mpsoc software execution at the transaction level. IEEE Des Test Comput 28(3):32–43
25. Popek GJ, Goldberg RP (1974) Formal requirements for virtualizable third generation architectures. Commun ACM 17(7):412–421
26. Rohou E, Williams K, Yuste D (2013) Vectorization technology to improve interpreter performance. ACM Trans Archit Code Optim (TACO) 9(4):1–22
27. Seznec A, Michaud P (2006) A case for (partially) tagged geometric history length branch prediction. J Instr Lev Parall 8:1–23
28. Sites RL, Chernoff A, Kirk MB, Marks MP, Robinson SG (1993) Binary translation. Commun ACM 36(2):69–81. doi:10.1145/151220.151227
29. Ung D, Cifuentes C (2000) Machine-adaptable dynamic binary translation. ACM SIGPLAN Not 35(7):41–51
30. Witchel E, Rosenblum M (1996) Embra: fast and flexible machine simulation. ACM SIGMETRICS Perform Eval Rev 24(1):68–79

Host-Compiled Simulation

19

Daniel Mueller-Gritschneder and Andreas Gerstlauer

Abstract

Virtual Prototypes (VPs), also known as virtual platforms, have been now widely adopted by industry as platforms for early software development, HW/SW coverification, performance analysis, and architecture exploration. Yet, rising design complexity, the need to test an increasing amount of software functionality as well as the verification of timing properties pose a growing challenge in the application of VPs. New approaches overcome the accuracy-speed bottleneck of today's virtual prototyping methods. These next-generation VPs are centered around ultra-fast host-compiled software models. Accuracy is obtained by advanced methods, which reconstruct the execution times of the software and model the timing behavior of the operating system, target processor, and memory system. It is shown that simulation speed can further be increased by abstract TLM-based communication models. This support of ultra-fast and accurate HW/SW cosimulation will be a key enabler for successfully developing tomorrows Multi-Processor System-on-Chip (MPSoC) platforms.

Acronyms

API	Application Programming Interface
CFG	Control-Flow Graph
HAL	Hardware Abstraction Layer
HW	Hardware
IPC	Inter-Process Communication

D. Mueller-Gritschneder (✉)
Department of Electrical and Computer Engineering, Technical University of Munich, Munich, Germany
e-mail: daniel.mueller@tum.de

A. Gerstlauer
Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA
e-mail: gerstl@ece.utexas.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_18

IR	Intermediate Representation
ISA	Instruction-Set Architecture
ISS	Instruction-Set Simulator
MPSoC	Multi-Processor System-on-Chip
OS	Operating System
SLDL	System-Level Description Language
TD	Temporal Decoupling
TLM	Transaction-Level Model
VP	Virtual Prototype
WCET	Worst-Case Execution Time

Contents

19.1 Introduction

Due to increased complexity of modern embedded and integrated systems, more and more design companies are adopting virtual prototyping methods. A Virtual Prototype (VP), also known as a virtual platform, is a computer model of a HW/SW system. In such a HW/SW system, *tasks* of an application are executed on one or more *target processors*, e.g., ARM cores. Tasks usually run on top of an Operating System (OS). The tasks can communicate and access memory and peripherals via communication fabrics, e.g., on-chip busses. Next to obtaining correct hardware with less iterations, VPs support early software development, performance analysis, HW/SW coverification, and architecture exploration.

Modern Multi-Processor System-on-Chip (MPSoC) platforms feature multiple hardware and software processors, where processors can each have multiple cores,

all communicating over an interconnection network, such as a hierarchy of busses. The large amount of functionality and timing properties that need to be validated for complex MPSoCs brings traditional VP approaches to their limits. New VPs are required, which raise the abstraction to significantly increase simulation speed. This is a challenging task, because high abstraction leads to a loss of timing information, penalizing simulation accuracy.

This gives rise to next-generation VPs, which are centered around host-compiled software models. Abstraction is applied at all layers of the system stack starting from the software level, including operating system and processor models, down to abstract communication models. Intelligent methods are applied to preserve simulation accuracy at ulta-high simulation speeds. These methods are independent of the used System-Level Description Language (SLDL). Yet, SystemC [1] has nowadays emerged as a quasi-standard for system-level modeling. Therefore, SystemC is used as the main SLDL to illustrate the modeling concepts throughout this chapter.

Next to ultra-high simulation speed, VPs based on host-compiled simulation also provide improved debug abilities. Both software and hardware events can be traced jointly and transparently as the simulation model describes both in one executable.

19.1.1 Traditional Virtual Prototype Simulation

The VP is simulated via an SLDL simulation kernel. The PC, which runs the simulation, is referred to as the *simulation host*. Naturally, it can have a different Instruction-Set Architecture (ISA) from the target processors. In discrete-event (compound adjective) simulation, the simulation kernel on the host advances the logical *simulation time*. To simulate concurrent behavior, simulation processes are sequentially executed based on *scheduling events* and the simulation time. Scheduling events suspend or resume simulation thread processes (*threads*) or activate method processes. Suspending and resuming the thread processes requires context switches, which can produce significant simulation overhead. This overhead reduces simulation speed, which measures how fast the simulation is performed in terms of the *physical time*.

Communication is usually modeled using abstract Transaction-Level Models (TLMs). TLMs center around memory-mapped communication but omit the detailed simulation of the bus protocol. In TLM, the bus interface is modeled by a TLM socket. A transaction is invoked by an *initiator* (master) module when calling a predefined transport function on its socket. The function is implemented at the *target* (slave) module. During simulation the initiator socket is bound to the target socket, and the respective transport function is called.

While TLMs have been successfully applied to model communication aspects, today's VPs usually model the computational part by emulating the software on Instruction-Set Simulators (ISSs), which are either inaccurate or slow. As such, the amount of functionality and timing properties that can be checked by traditional VPs remains limited by their simulation speed. The low speed results from the

high number of scheduling events created by the traditional computation and communication model.

19.1.2 Next-Generation Virtual Prototypes

Simulation speed can be increased by reducing the number of scheduling events. One possibility is to raise the level of abstraction and thus lower the level of detail in the simulation. Abstractions can increase simulation speed significantly but may decrease accuracy due to loss of timing information or missing synchronization events as discussed in Sect. 19.1.3.

Next-generation VPs are aimed at overcoming these challenges by using intelligent modeling approaches to preserve simulation accuracy. The abstraction is raised by source-level simulation of software as an advanced method for software performance analysis. Instead of emulating the software program with an ISS of the target processor at the binary level, the source code of the software is directly annotated with timing or other information. The annotated source code can be directly compiled and executed on the simulation host machine, which leads to a huge gain in simulation speed compared to ISS simulation. However, this requires access to the source code by the user, which might not be available especially for library functions. This is a limitation of such approaches, which can be partly overcome, e.g., by profiling library functions beforehand. Additionally, sophisticated methods are required to annotate potentially expensive and slow cosimulation models for any dynamic low-level target behavior, such as stack or cache behavior, that cannot be easily or accurately estimated through static analysis

Pure source-level simulation approaches focus on emulating stand-alone application behavior only. However, interferences among multiple tasks running on a processor as well as hardware/software interactions through interrupt handling chains and memory and cache hierarchies can have a large influence on overall software behavior. As such, OS and processor-level effects can contribute significantly to overall model accuracy, while also carrying a large simulation overhead in traditional solutions. So-called host-compiled simulation approaches therefore extend pure source-level models to encapsulate back-annotated application code with abstract, high-level, and lightweight models of OSs and processors. This is aimed at providing a complete, fast, and accurate simulation of source-level software running in its emulated execution environment.

Additionally, embedded and integrated systems are composed out of many communicating components as we move toward embedded multi-core processors. The simulation of communication events can quickly become the bottleneck in system simulation. Next-generation VPs tackle this challenge by providing abstract communication models. Special care must be taken in such abstract models to capture the effect of conflicts due to concurrent accesses on shared resources. Different methods are addressed, which can model the effect of arbitration, e.g., by retroactive correction of the timing behavior. This correction is usually performed by a central timing manager.

19.1.3 Temporal Decoupling

Overall, any discrete-event simulation of asynchronous interactions among concurrent system components will always come with a fundamental speed and accuracy trade-off. Concurrency is simulated by switching between execution of the simulation processes at scheduled simulation events. With increasing amount of such scheduling events, simulation speed drops due to the overhead caused by the involved context switches.

Naturally, simulation speed can be increased by reducing the number of scheduling events. This can be achieved by raising the level of abstraction in the simulation model. A less detailed simulation usually leads to fewer scheduling events. However, a simulation at coarser granularity also leads to timing information being potentially lost. Maintaining a coarser timing granularity results in a Temporal Decoupling (TD) of simulation processes. TD lets simulation processes run ahead of the logical simulation time up to a given *time quantum*, which increases the simulated timing granularity and decreases the number of scheduling events. The logical simulation time of the kernel is referred to as *global simulation time*. By contrast, components keep track of their time using a *local simulation time*, which is usually defined as an offset to the global simulation time.

For HW/SW systems, TD increases simulation speed but may decrease accuracy due to out-of-order accesses to or wrong prediction of conflicts on shared resources. Specifically, TD may decrease accuracy due to incoming events being captured too late, and also in terms of outgoing events being produced too early. This is illustrated for a scenario with two concurrent active threads in Fig. 19.1. Without TD, as shown in the upper half of the figure, Thread 2 writes the value of b to the shared target before it is read by Thread 1. Additionally, the shared target can arbitrate accesses. The writing process of b is ongoing when the read access is performed, which adds additional delay on the read access of Thread 1. In contrast, temporal decoupling leads to out-of-order accesses to b as shown in the lower half of the figure. Additionally, the conflicting access by Thread 2 cannot be predicted by the shared resource, and Thread 1 sees the write to b by Thread 2 only at a much later (local) time. Similar problems of events being recognized too late arise when modeling active threads that can be interrupted or preempted by external sources.

TD methods can be classified as optimistic or conservative. Optimistic approaches aggressively execute a model under temporal decoupling. Inaccuracies due to out-of-order execution are either tolerated or corrected for at a later point in simulation. Note that optimistic approaches do not guarantee an accurate order of events and interactions unless correction using a full rollback is possible in the simulator. As the name suggests, conservative approaches, by contrast, always maintain the correct order of events and interactions. They only apply temporal decoupling as long as it can be guaranteed that results do not depend on the order of the respective events. Both methods can benefit from intelligent compile-time or run-time usage of system knowledge. Accuracy of optimistic approaches can be improved by using system knowledge to perform retroactive timing corrections. Conservative approaches can increase their simulation speed by dynamically

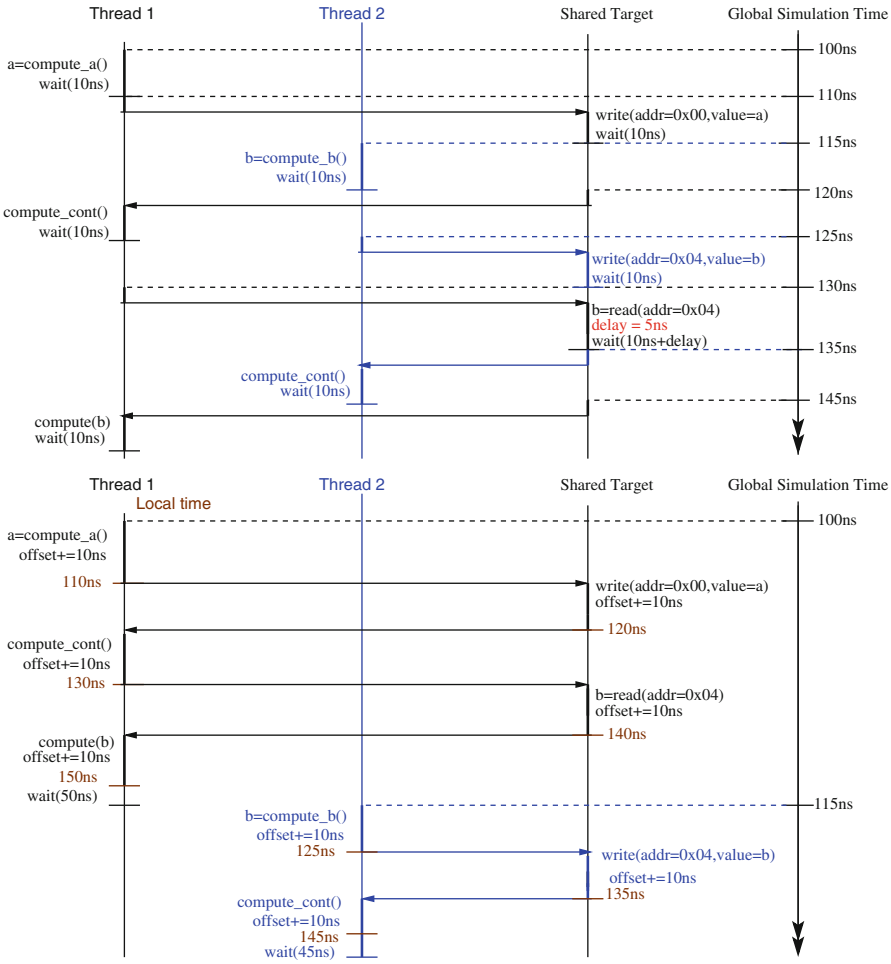


Fig. 19.1 Simulation without/with Temporal Decoupling (TD)

adjusting their local time quantum using system knowledge to perform prediction of possible future event interactions. In Sects. 19.2 and 19.4, we will show in detail how accuracy and speed can be improved by such intelligent TD modeling approaches.

19.2 Source-Level Software Simulation

Traditional virtual platforms simulate software execution at a detailed instruction level. This includes both a functional as well as, optionally, a timing model. Such low-level ISSs can be very accurate, especially when combined with a cycle-level microarchitecture model, but they also tend to be very slow, especially when cosimulating multiple processor or cores in a full-system context. Functional simulation

speed can be significantly improved by statically translating instructions (and caching translated results, see the ▶ Chap. 18, "Multiprocessor System-on-Chip Prototyping Using Dynamic Binary Translation") instead of dynamically interpreting them. Timing models can be accelerated by executing them in FPGAs or other dedicated hardware platforms [6]. Nevertheless, simulation speed, particularly when requiring accurate timing, remains a major concern.

Source-level approaches are aimed at improving the speed of both functional and timing simulations. Computation is modeled at the source or Intermediate Representation (IR) level, which allows a purely functional model to be natively compiled and executed on a host without having to emulate the functionality of a target ISA. For fast timing simulation, source-level methods employ a hybrid approach that combines the functional simulation with an abstract, statically derived timing model at much coarser program block granularity. This is similar to static Worst-Case Execution Time (WCET) estimation. However, a key challenge is enumeration of possible program paths when performing such static analysis at the whole program level. Source-level approaches avoid or simplify this problem by constructing a static timing model at finer block-level granularity and driving this model with block sequences of program paths encountered in the actual functional simulation. In practice, this is often done by simply back-annotating the timing model directly into the functional source or IR code. Nevertheless, models can be separated, and coarse-grain timing models can equally be combined with fast functional ISS models instead (see also ▶ Chaps. 20, "Precise Software Timing Simulation Considering Execution Contexts" and ▶ 21, "Timing Models for Fast Embedded Software Performance Analysis").

A remaining challenge is that due to pipeline, cache, and other effects, the timing of a program block is not statically fixed but generally depends on the dynamic machine state and hence previous program history. Traditional ISSs simulate detailed interactions with machine state for each encountered instruction. WCET approaches have to derive conservative bounds based on possible program history. By contrast, source-level approaches operate at an intermediate level. The functional simulation allows actual history and state-dependent effects to be accurately tracked. At the same time, only the relevant state is dynamically simulated while statically abstracting as many effects as possible. During static pre-characterization, blocks are analyzed on a target reference model under different possible conditions. Static timing numbers can then be selected based on the actual context encountered in simulation. This can provide accurate timing without replaying the same instruction sequence on the slow timing reference each time the block is executed. However, this is only possible as long as the possible state space affecting block timing is small. For dynamic structures with deep history, such as caches or branch predictors, detailed simulation models that dynamically adjust pre-characterized block timing can be included. Alternatively, a static average- or worst-case analysis can be applied. Ultimately, the amount of static analysis versus dynamic simulation overhead thereby determines the speed and accuracy trade-off in different source-level models.

A typical flow for generating a source-level timing simulation model is shown in Fig. 19.2. There are generally three stages in such a framework: (1) binary

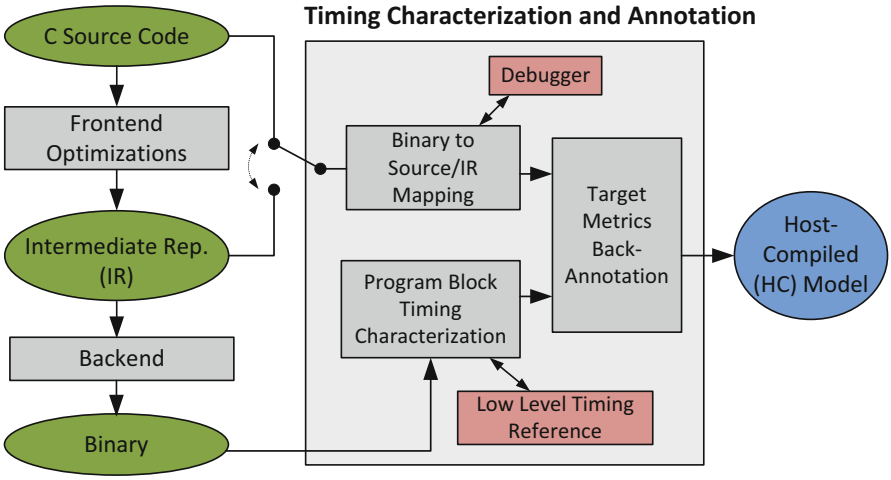


Fig. 19.2 Source-level timing characterization and back-annotation

to source/IR mapping, (2) program block timing characterization, and (3) target metrics back-annotation. During the mapping stage, a relationship between source-/IR-level code and binary need to be established. This usually includes an accurate Control-Flow Graph (CFG) mapping to provide insertion points of target profiling metrics. Additionally, memory accesses also need to be reconstructed in order to account for cache effects. For timing characterization, target metrics are extracted at the machine level at a certain granularity as a one-time effort. Finally, execution statistics are back-annotated into the source/IR model based on the previously determined relationship. The back-annotated source code can then be directly compiled and executed on the simulation host machine for fast and accurate software simulation.

19.2.1 Binary to Source Mapping

The first step in the timing back-annotation flow is to establish a matching between the CFGs of the target binary and source-level code, which is aimed at ultimately allowing target metrics to be annotated back into the source-level or IR code at correct insertion points.

The earliest works estimate and annotate target performance metrics purely based on source code analysis [4, 14]. Control flow and operation information are extracted directly at source level, which are further fed into abstract performance estimators to calculate the corresponding target metrics. These approaches avoid the mapping issues and provide a fast and approximate profiling strategy for early stage design space exploration. However, without consulting the corresponding target binary, such techniques cannot provide precise estimations compared to detailed characterization of binary blocks on a cycle-level reference model.

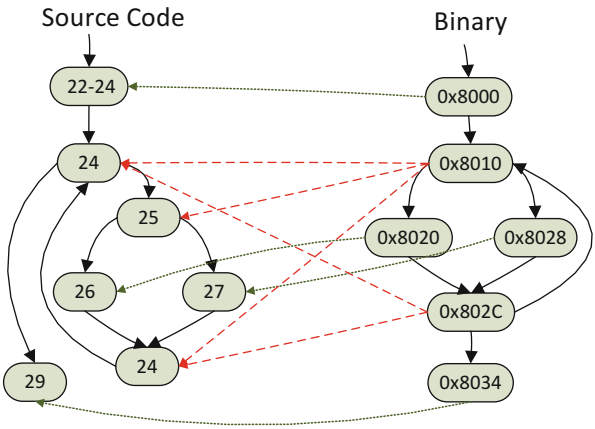


Fig. 19.3 Matching between source- and binary-level control flow

The challenge when working at the binary level is that, due to aggressive compiler optimization, CFGs can be significantly changed when source code is transformed into a binary implementation (Fig. 19.3). Thus, target metrics obtained for each binary block need to be annotated to a matching source code block such that numbers are guaranteed to be accounted for correctly when accumulated during subsequent source-level simulation. Early approaches relied on debug information alone to perform such a mapping [45]. However, debug information provided by standard compilers is often unreliable or ambiguous. For example, debug information often contains no or several source references for the same binary block. Each of these entries describes a potential relation between a binary and source-level basic blocks from which a unique mapping still needs to be determined, i.e., the debug information has to be subject to further analysis steps. These ambiguities and CFG mismatches are the main issues to be resolved to establish a valid and accurate mapping.

When targeting back-annotation at the source level, binary to source matching typically relies on complex structural analysis combined with the use of IR level and debug information to establish a mapping between the target binary and source code [24]. Structural analysis is performed on both source level and binary code to extract loop and control flow dependency characteristics. Along with debug information, these structural properties are then used as matching criteria to establish a tentative CFG mapping. In case of heavily optimized CFGs, advanced approaches annotate an additional IR level [43] or binary path simulation model [39, 41] to dynamically cosimulate, track, and reconstruct the actual binary execution flow and its corresponding accumulated timing. Working at source level benefits from better readability and convenience with respect to manual adjustments or analysis of the simulation model. However, it is usually hard to systematically handle the full range of compiler optimization for general CFG mapping.

Other works address these issues by performing back-annotation and simulation at the IR level. Working at the IR allows typical front-end compiler optimizations to be taken into account, where the IR provides a much closer representation of the final control flow. This simplifies the matching problem, i.e., improves accuracy with little to no penalty in execution speed. Early work [45] only used debug information to perform the mapping, which is more reliable when working at the IR level. Nevertheless, even IR and binary control flows do not always match cleanly due to aggressive compiler backend optimizations. In these cases, similar to advanced source-level approaches, a path tracking model that replicates the CFG of the target binary can be extracted during backend code generation for cosimulation with the IR [3]. However, this adds simulation overhead and requires detailed backend compiler and/or target ISA information to be available, making the approach dependent on the estimation target. Alternatively, a mapping can be established by a simplified structural analysis of both IR and binary CFGs. A general and fully retargetable approach is proposed in [5], where a synchronized depth-first traversal of both CFG is performed to identify legal matches based on a control flow representation using both loop and branch nesting levels. In addition, debug information is consulted when multiple equally likely matches are possible.

19.2.2 Memory Trace Reconstruction

Caches can have a large effect on overall performance estimation. At the same time, cache behavior is highly dynamic and strongly depends on the actual sequence of memory accesses made by the application, which cannot be fully determined statically during program block characterization. Some approaches employ an approximate solution by annotating a statistically calculated average or statically estimated worst-case delay for each memory access [14, 18]. Otherwise, memory accesses need to be reconstructed during the back-annotation stage with information from both binary and source code [21, 27, 44]. The source or IR code is thereby annotated with accurate information about the type and address of each memory access made by the binary code. Alternatively, approaches that already reconstruct a binary cosimulation model for path tracking purposes can equally annotate this model with memory access tracking code obtained from de-compiling the binary [42]. In either case, back-annotated memory accesses can then feed an abstract, dynamic cache simulation model that determines additional cache miss and memory delay penalties to be included during source-level timing simulation (see Sect. 19.3.3).

Memory access trace reconstruction can be generally decomposed into three categories: (1) accesses to static and global variables, (2) accesses to stack data, and (3) accesses to the heap. To reconstruct the addresses of static and global data accesses, their base addresses and, in case of non-scalars, their access offsets are required. Base address information of global data can easily be obtained from the symbol table of the target binary, while access offsets are extracted from analysis of IR or binary code. A key observation is that, with proper translation of primitive data types, access offsets in the IR are the same as in the target binary, while only base

addresses differ. Hence, IR-based approaches can directly obtain such information. By contrast, reconstructing accurate addresses at the source level requires falling back to IR or binary analysis.

Different from the global data, stack and heap accesses are more complicated to back-annotate. Their base addresses change dynamically depending on the local and global execution context. Tracking such memory accesses requires reconstructing the target stack/heap layout as well as the dynamic status of the stack pointer and heap manager during program execution. Thus, abstracted models for stack pointers and heap allocation are usually inserted into source or IR-level simulations to capture and track such information dynamically. Together with access offsets extracted from IR or source code, accurate target stack/heap accesses can then be reconstructed during simulation time.

19.2.3 Block-Level Timing Characterization

The core step in the back-annotation process is the characterization of block-specific target metrics. As mentioned above, accurate characterization is complicated by the fact that target metrics of a program block can be significantly affected by the dynamic machine state and previous program history. In general, the performance metrics for a code block are determined by its internal execution paths as well as the path history of code that has previously executed (Fig. 19.4).

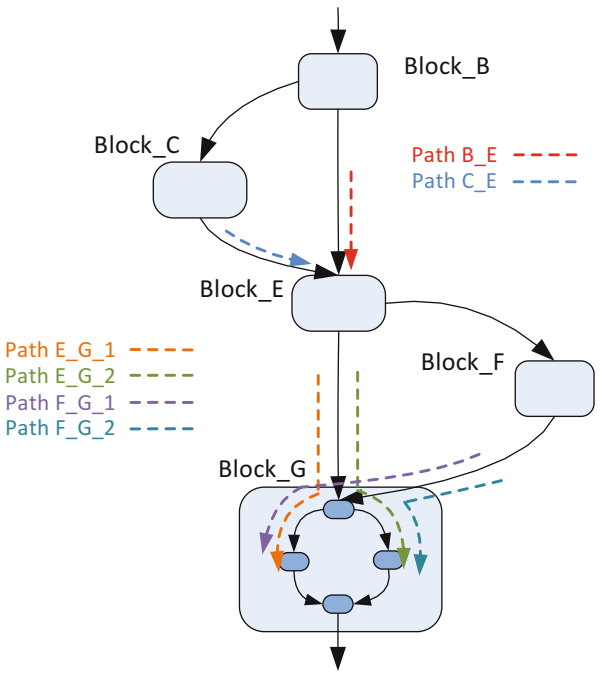


Fig. 19.4 Path dependency of block-level timing characterization

Overall speed and accuracy are determined by the granularity of code blocks at which characterization (and ultimately back-annotation are performed). Approaches that operate at a coarse function level [4] have to estimate average or worst-case behavior across all dynamic execution paths taken within each function body as determined, for example, by a previous profiling run. By contrast, solutions that annotate timing at the individual statement level [22] suffer from unnecessary characterization and simulation overhead. Most existing approaches instead work at a basic block level. Since there is only a single path through a basic block, the assumption is that its baseline timing can be accurately represented by a single, statically characterized number.

Machine state and execution history can, however, still significantly affect a block's timing. Such effects can be estimated by employing a WCET analysis for binary timing characterization, which provides an upper bound on the execution time for each individual basic block either alone or within its larger execution context [41]. In most cases, however, basic block timing is characterized through cycle-accurate simulation on a detailed microarchitecture reference model. Pipeline history effects are taken into account by characterizing each block in sequence with possible predecessors, such that variations in execution context are accurately accounted for. The characterization overhead thereby exponentially increases with the number of possible predecessors and depth of considered execution path history. Depending on the binary block length and machine pipeline depth, a maximum number of predecessors that can possibly affect dependencies until they are guaranteed to have percolated through the pipeline can be dynamically determined for each block and path [18]. Alternatively, blocks can simply be characterized through pairwise execution with all of their immediate predecessors, which has been shown to provide a good trade-off between accuracy and estimation complexity [5].

19.2.4 Back-Annotation

The final step in generating a source-level software simulation model is back-annotation of characterized timing metrics using previously obtained mapping information. Metrics gathered during the characterization step are usually recorded in a mapping table, which is then used for directing the annotation of target metrics into the IR or source code at correct insertion points.

For a single annotation unit, there are often multiple performance metrics accounting for path-dependent timing effects. In order to be able to pick up the correct set of metrics during simulation, the back-annotation process will usually insert extra data structures to record essential execution history, such as the dynamic predecessor of currently executing basic block. In this way, the back-annotated model can reconstruct the binary execution flow and properly accumulate block execution time along with the annotated points.

To account for dynamic execution characteristics that depend on complex history behavior, such as branch predictors [8] and caches (see Sect. 19.3.3), the source code is further augmented with calls to dynamic simulation models of such (micro-)

architectural structures. In case of caches, this includes annotating the code with previously reconstructed memory and cache access information. During simulation, delays can be adjusted according to the corresponding outcomes from such models. For this purpose, blocks are usually characterized assuming ideal conditions (such as always-correct branch prediction or perfect caches), where dynamically determined penalties are added during simulation. This approach is feasible for simpler in-order processors. By contrast, dynamic tracking of complex interactions between pipelines and other structures in out-of-order processors requires a significantly more involved characterization and back-annotation [26]. In all cases, the choice of annotating static estimates or dynamic simulation models, which incur additional, in some cases, significant simulation overhead, enables generation of different models with varying trade-off between simulation speed and accuracy. Furthermore, such source-level simulation approaches can be extended beyond timing to back-annotation of other performance, energy, reliability, power, and thermal (PERPT) metrics [5, 9, 17, 47].

19.3 Host-Compiled OS and Processor Modeling

As described previously, host-compiled simulators extend pure source-level approaches with fast yet accurate models of the complete software execution environment. Figure 19.5 shows a typical layered organization of a host-compiled simulation model [30, 35, 38]. Individual source-level application models that are annotated with timing and other metrics as described in Sect. 19.2 are converted

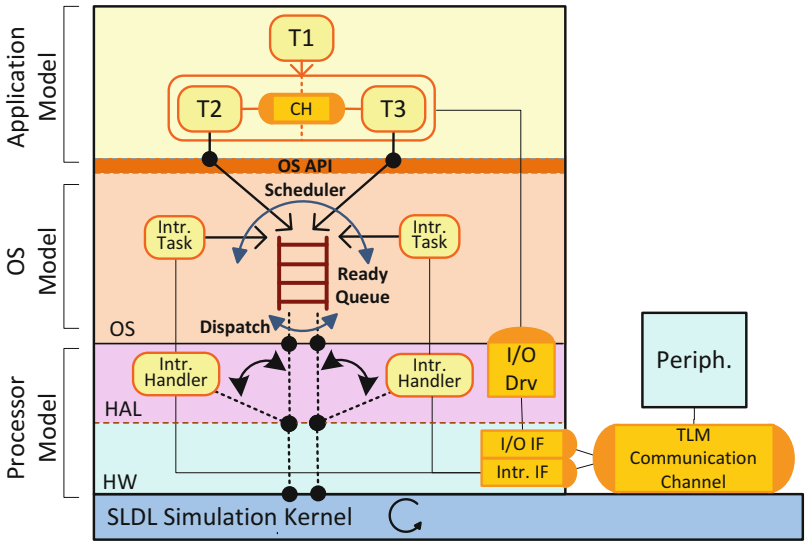


Fig. 19.5 Host-compiled simulation model

into tasks running on top of an abstract, canonical OS Application Programming Interface (API). Tasks are grouped and encapsulated according to a given partitioning to model the multi-threaded application mix running on each processor of an overall MPSoC. Within each processor, an OS model then provides an implementation of the OS API to manage tasks and replicate a specific single- or multi-core scheduling strategy. The OS model itself sits on top of models of the firmware and drivers forming a Hardware Abstraction Layer (HAL). An underlying Hardware (HW) layer in turn provides interfaces to external TLMs of the communication infrastructure (Sect. 19.4). Finally, the complete processor model is integrated and cosimulated with other system components on top of an SLDL. The SLDL simulation kernel thereby provides the basic concurrency and synchronization services for OS, processor, and system modeling.

19.3.1 OS Modeling

An OS model generally emulates scheduling and interleaving of multiple tasks on one or more cores [11, 12, 16, 23, 28, 31, 46]. It maintains and manages tasks in a set of internal queues similar to real operating systems. In contrast to porting and paravirtualizing a real OS to run on top of the modeled HAL, a lightweight OS model can provide an accurate emulation of real OS behavior with little to no overhead [35]. Tasks are modeled as parallel simulation threads on top of the underlying SLDL kernel. The OS model then provides a thin wrapper around basic SLDL event handling and time management primitives, where SLDL calls for advancing simulation time, event notification, and wakeup in the application model are replaced with calls to corresponding OS API methods. This allows the OS model to suspend, dispatch, and release tasks as necessary on every possible scheduling event, i.e., whenever there is a potential change in task states. An OS model will typically also provide a library of higher-level channels built around basic OS and SLDL primitives to emulate standard application-level Inter-Process Communication (IPC) mechanisms.

Figure 19.6 shows an example trace of two tasks T_0 and T_1 running on top of an OS model emulating a time slice-based round-robin scheduling policy on a single core [30]. Source-level execution times of tasks are modeled as calls to wait-for-time methods in the OS API. On each such call, the OS model will advance the simulated time in the underlying SLDL kernel but will also check whether the time slice is expired and switch tasks if this is the case. In order to simulate such a context switch, the OS model suspends and releases tasks on events associated with each task thread at the SLDL level. Overall, the OS model ensures that at any simulated time, only one task is active in the simulation. Note that this is different from scheduling performed in the SLDL kernel itself. Depending on available host resources, the SLDL kernel may serialize simulation threads in physical time. By contrast, the OS model serializes tasks in the simulated world, i.e., in logical time.

Within an isolated set of tasks on a core, this approach allows OS models to accurately replicate software timing behavior for arbitrary scheduling policies.

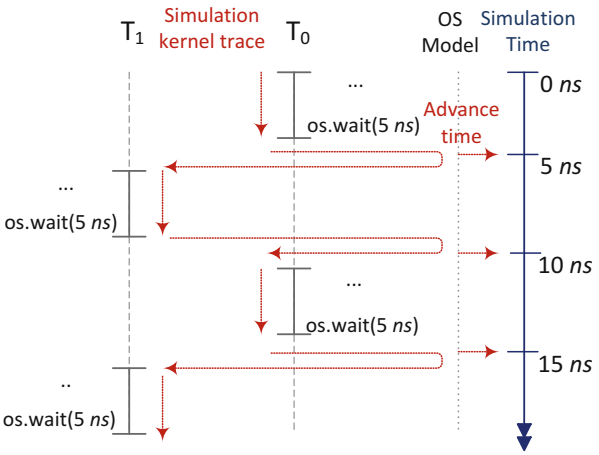


Fig. 19.6 Example of OS model trace

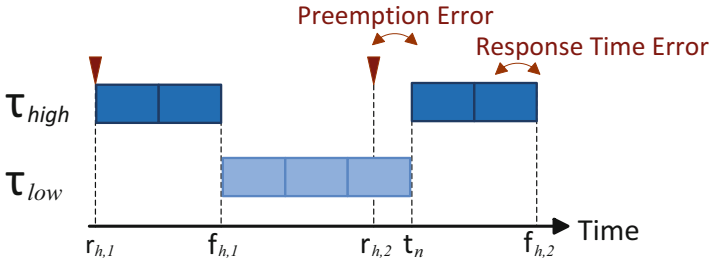


Fig. 19.7 Inherent preemption inaccuracies in discrete OS models

However, the discrete nature of such models introduces inherent inaccuracies in the presence of asynchronous scheduling events, such as task releases triggered by external interrupts or by events originating on other cores. Since the OS model advances (simulated) time only in discrete steps, it will not be able to react to such events immediately. Figure 19.7 shows an example of a low-priority task τ_{low} being preempted by a high-priority task τ_{high} triggered externally. In reality, the high-priority task is released at time $r_{h,2}$. In the simulation, however, the OS model is not able to perform the corresponding task switch until the next simulation step is reached at time t_n. This results in a corresponding preemption and response time error for both tasks (with τ_{low} potentially finishing too early).

As shown in the example of Fig. 19.7, the preemption error is generally upper bounded by the maximum timing granularity. By contrast, it can be shown that response time errors can potentially become much larger than the time steps themselves [33]. This is, for example, the case if τ_{low} in Fig. 19.7 finishes too early but should have been preempted and delayed by a very long running τ_{high}. This can be a serious problem for evaluation of real-time system guarantees. Adjusting

the timing granularity does not generally help to improve the maximum simulation error. Nevertheless, decreasing the granularity will reduce the likelihood of such large errors occurring, i.e., will improve average simulation accuracy.

At the same time, the timing granularity also influences simulation speed. A fine granularity allows the model to react quickly but increases the number of time steps, context switches and hence overhead in the simulator. Several approaches have been proposed to overcome this general trade-off and provide a fast coarse-grain simulation while maintaining high accuracy. Existing approaches are either optimistic [37] or conservative [32]. In optimistic solutions, a lower-priority task is speculatively simulated at maximum granularity assuming no preemption will occur. If a preemption occurs while the task is running, the higher-priority task is released concurrently at its correct time. In parallel, all disturbing influences are recorded and later used to correct the finish time of the low-priority task(s). Such an approach has also been used to model preemptive behavior in other contexts, such as in TLMs of busses with priority-based arbitration [36] (see also Sect. 19.4.2). Note that unless a full rollback is possible in the simulator, optimistic approaches cannot guarantee an accurate order of all task events and interactions, such as shared variable accesses. By contrast, in conservative approaches, at any scheduling event, the closest possible preemption point is predicted to select a maximum granularity not larger than that. If no prediction is possible, the model falls back onto a fine default granularity or a kernel mechanism that allows for coarse time advances with asynchronous interruptions by known external events. Conservative approaches, by their nature, always maintain the correct task order. In both optimistic and conservative approaches, the OS model will automatically, dynamically, and optimally accumulate or divide application-defined task delays to match the desired granularity. This allows the model to internally define a granularity that is independent from the granularity of the source-level timing annotations. Furthermore, both types of approaches are able to completely avoid preemption errors and associated issues with providing worst-case guarantees.

19.3.2 Processor Modeling

Host-compiled processor models extend OS models with accurate representations of drivers, interrupt handling chains, and integrated hardware components, such as caches and TLM bus interfaces [2, 10, 15, 38]. Specifically, accurate models of interrupt handling effects can contribute significantly to overall timing behavior and hence accuracy [35].

The software side of interrupt handling chains is typically modeled as special, high-priority interrupt handler tasks within the OS model [46]. On the hardware side, models of external generic interrupt controllers (GICs) interact with interrupt logic in the processor model's hardware layer. The OS model is notified to suspend the currently running task and switch to a handler whenever an interrupt for a specific core is detected. At that point, the handler becomes a regular OS task,

which can in turn notify and release other interrupt or user tasks. By back-annotating interrupt handlers and tasks with appropriate timing estimates, an accurate model of interrupt handling delays and their performance impact can be constructed.

An example trace for a complete host-compiled simulation of two task sets with three tasks each running on a dual-core platform is shown in Fig. 19.8 [30]. Task sets are mapped to run on separate cores, and the highest priority tasks are modeled as periodic. All interrupts are assigned to Core_0. The trace shows a conservative OS model using dynamic prediction of preemptions. The model is in a fine-grain fallback mode whenever there is a higher-priority task or handler waiting for an unpredictable external event. In all other cases, the model switches to a predictive mode using accumulation of delays. Note that high-priority interrupt handlers and tasks are only considered for determining the mode if any schedulable tasks is waiting for the interrupt. This allows the model to remain in predictive mode for the majority of time. Handlers and tasks themselves can experience large errors during those times. However, under the assumption that they are generally short and given that no regular task can be waiting, accuracy losses will be small.

When applied to simulation of multi-threaded, software-only Posix task sets on a single-, dual-, and quad-core ARM-Linux platform, results show that host-compiled OS and processor models can achieve average simulation speeds of 3,500 MIPS with less than 0.5% error in task response times [35]. When integrating processor models into a SystemC-based virtual platform of a complete audio/video MPSoC, more than 99% accuracy in frame delays is maintained. For some cases, up to 50% of the simulated delays and hence accuracy is attributed to accurately modeling the Linux interrupt handling overhead. Simulation speeds, however, drop to 1,400 MIPS. This is due to the additional overhead for cosimulation of HW/SW interactions through the communication infrastructure. Methods for improving performance of such communication models will be discussed in Sect. 19.4.

19.3.3 Cache Modeling

Next to external communication and synchronization interfaces, a host-compiled processor simulator will generally incorporate timing models for other dynamic aspects of the hardware architecture. Specifically, timing effects of caches and memory hierarchies are hard to capture accurately as part of a static source-level back-annotation. Hit/miss rates and associated delay penalties depend heavily on the execution history and the specific task interactions seen by the processor. To accurately model such dynamic effects, a behavioral cache simulation can be included [25, 27, 42, 44].

As described in Sect. 19.2, the source level can be annotated to re-create accurate memory access traces during simulation. Such task-by-task traces can in turn drive an abstract cache model that tracks history and hit/miss behavior for each access. Resulting penalties can then be used to dynamically update source-level timing annotations. Note that cache models only need to track the cache state in terms of line occupancy. The data itself is natively handled within the simulation host.

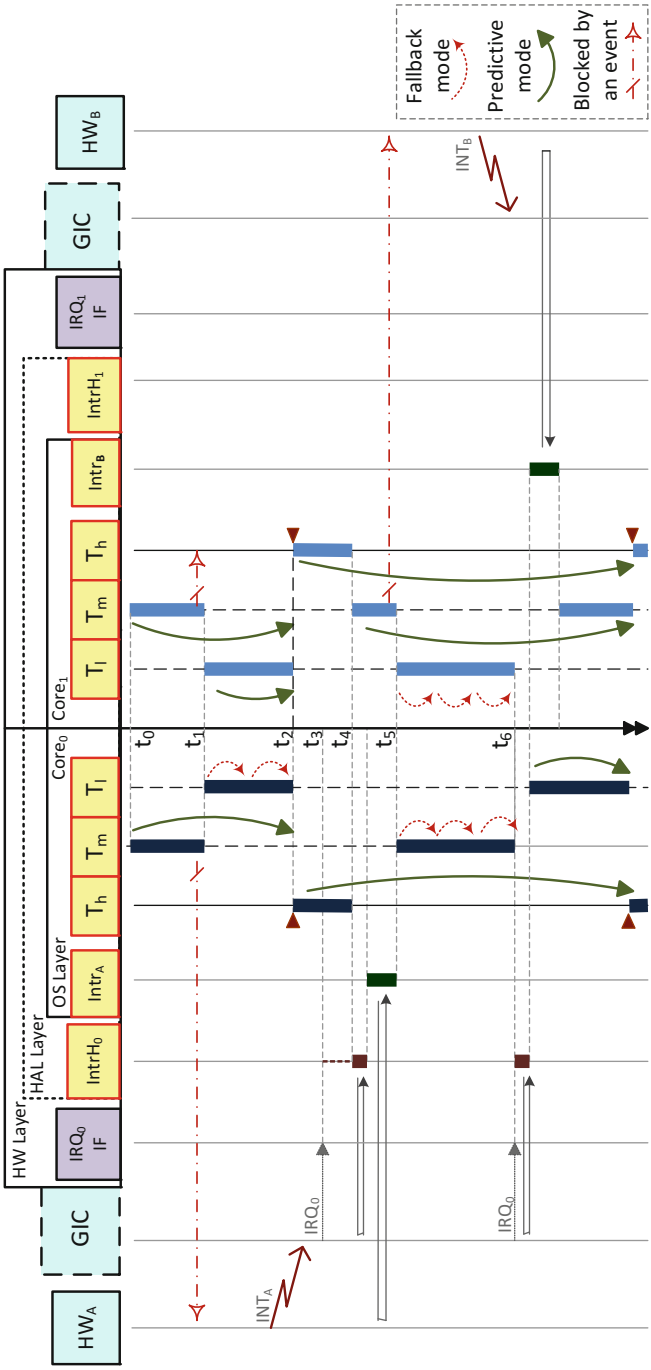


Fig. 19.8 Host-compiled simulation trace

When combined with an OS model, such an approach allows for accurate modeling of cache pollution and interference among different tasks. A particular challenge emerges, however, when multiple cores can interfere through a shared cache. A cache model can accurately track shared state, including coherency effects across multiple cache levels, as long as individual core models issue cache accesses in the correct global order. As mentioned above (Sect. 19.3.2), this is generally not the case in a coarse-grain, temporally decoupled simulation. Cores may produce outgoing events ahead of each other, and, as a result, multiple cores may commit their accesses to the cache globally out-of-order. At the same time, from a speed perspective, it is not feasible to decrease granularity to a point where each memory access is synchronized to the correct global time.

Several solutions have been proposed to tackle this issue and provide a fast yet accurate multi-core out-of-order cache (MOOC) simulation in the presence of temporal decoupling [34, 40]. The general approach is to first collect individual accesses from each core including accurate local time stamps. Later, once a certain threshold is reached, accesses are reordered and committed to the cache in their globally correct sequence. Figure 19.9 illustrates this concept [34]. In this approach, both cores first send accesses to a core-specific list maintained in the cache model. After each time advance, cores notify the cache to synchronize and commit all accesses collected up to the current time. It is thereby guaranteed that all other cores have advanced and produced events up to at least the same time.

An added complication are task preemptions [30]. Since cores and tasks can run ahead of time, a task may generate accesses that would otherwise not have

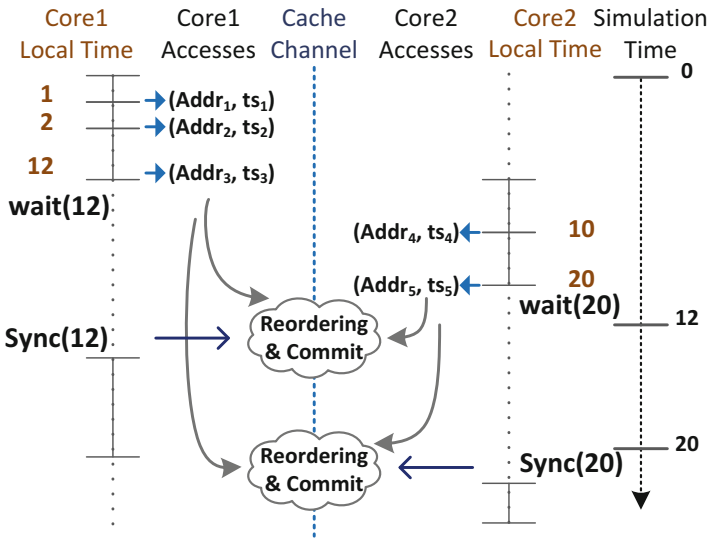


Fig. 19.9 Multi-core out-of-order cache simulation trace

been issued until after a possible preemption is completed. This requires access reordering to be tightly integrated with the OS model. By maintaining task-specific access lists in the OS model instead of the cache, the OS can adjust remaining time stamps by the duration of the preemption whenever such a preemption occurs. Overall, such an approach can maintain 100% accuracy of cache accesses at the speed of a fully decoupled simulation.

In other approaches, the cache model is moved outside of the processor to become part of the TLM backplane itself [40]. In this case, the cache is accessed via regular bus transactions, and all of the reordering is relegated to a so-called quantum giver within a temporally decoupled TLM simulation (see Sect. 19.4.3). Note that this still requires OS model support to generate accurate transaction time stamps in the presence of preemptions. Similar reordering techniques can then also be applied to other shared resources, such as busses, as will be shown in the following sections.

19.4 TLM Communication for Host-Compiled Simulation

Embedded and integrated systems are comprised of many communicating components as we move toward embedded multi-core processors. Fast simulation requires advanced communication models at transaction level. Usually, scheduling events of the simulation kernel are closely coupled to the communication events. For high-speed virtual prototyping, usually the blocking transaction style, called loosely-timed TLM in SystemC [1], is applied. Blocking transactions can be synchronized to the global simulation time at each accessed module. As many communication resources such as busses or target (slave) modules are shared between initiator (masters) modules, accurate models additionally schedule an arbitration event at each arbitration cycle.

Novel works have shown that these requirements can be usually relaxed to improve simulation speed. These works either raise the abstraction of the communication, e.g., a single simulated block transaction represents a set of bus transactions performed by the HW/SW system, or apply TD. TD implies that initiators perform accesses, which are located in the future with respect to the current global simulation time. This leads to several challenges, as discussed in Sect. 19.1.3. An overview of selected communication models is given in the following.

19.4.1 TD with No Conflict Handling

The TLM-2.0 standard offers the Quantum Keeper. The TLM-2.0 Quantum Keeper provides a global upper bound to the local time offset. The Quantum Keeper is easily applicable to realize temporal decoupling. It offers no standard way to handle data dependencies or resource conflicts. Shared variables have to be protected by additional synchronization methods. Figure 19.10a shows a message diagram for an example. We assume blocking TLM communication style indicated by the

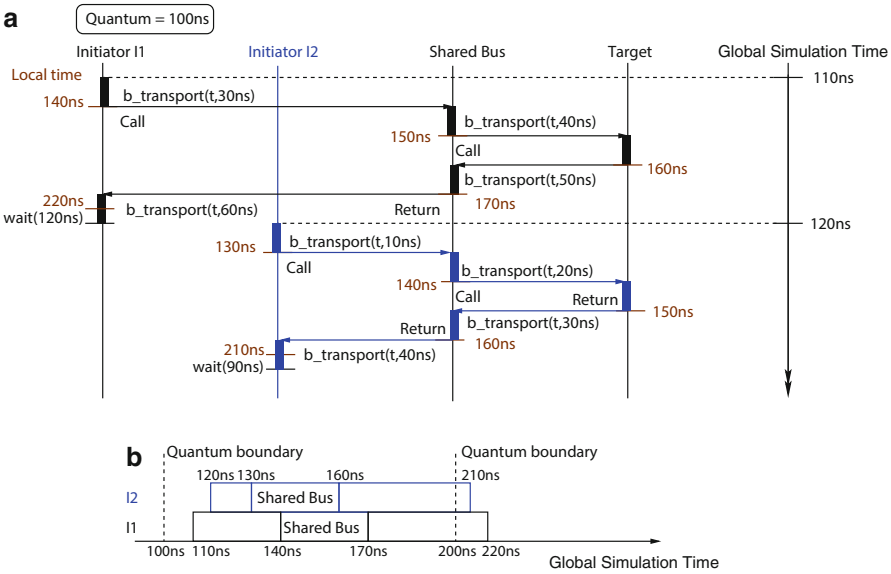


Fig. 19.10 Temporal decoupling with Quantum Keeper

b_transport function. The b_transport function propagates as its argument of the offset between the local simulation time and the global simulation time. Both initiators are only executed once per quantum until their local time exceeds the next quantum boundary. The communication is out-of-order. Transaction of initiator I1 to the shared bus starts at 140 ns, yet it is executed before the transaction of I2 starting at 130 ns. Additionally, the transactions concurently access the shared bus as shown in Fig. 19.10b, which would lead to arbitration. Yet, I1 can finish its transaction without conflict delay because the transaction of I2 was not yet known at the shared bus. Simulation speed is highest, but communication timing is optimistic, and out-of-order accesses may lead to incorrect simulation results. Thus, several methods for using system knowledge for improving accuracy in TLM communication using TD were proposed, which are presented in the following.

19.4.2 TD with Conflict Handling at Transaction Boundaries

In [29, 36], the additional delay due to resource conflicts are resolved at the transaction boundaries. At the start of a transaction, the communication state is inspected in [36]. If a higher-priority transaction is ongoing, the end time of the considered transaction is computed accordingly. Yet, still an optimistic end time is computed at the beginning of a transaction because future conflicting transactions are not considered. When the end time of the transaction is reached, additional delay due to other conflicting transaction is retroactively added. In the case of [36], another

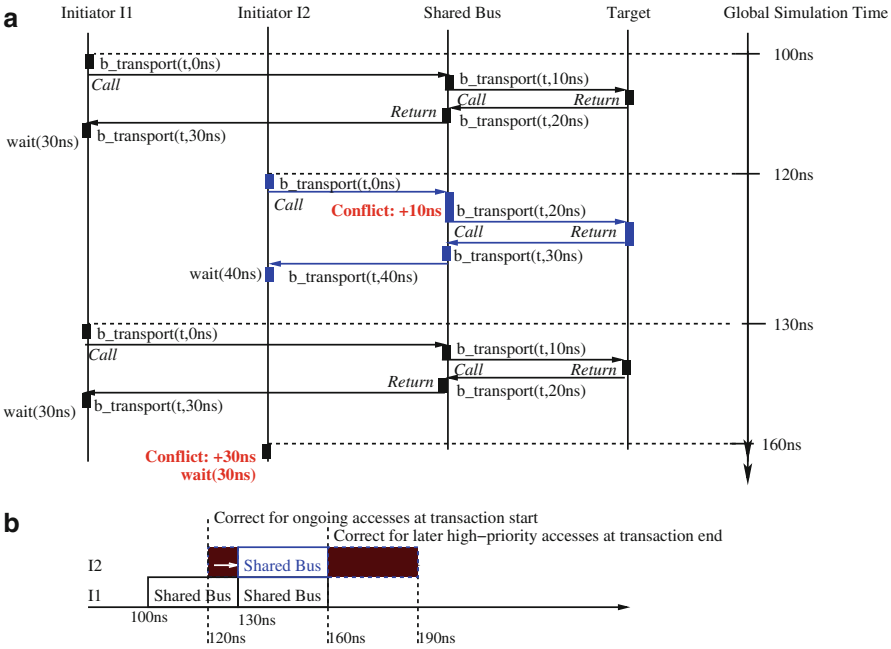


Fig. 19.11 Handling conflicts at transaction boundaries

wait is issued to account for the additional delay, but intelligent event re-notification could also be applied. The method needs minimally one context switch (single call to wait()) per transaction. An example is shown in Fig. 19.11 with I1 having higher priority on the shared bus. The transaction of I2 is delayed during execution due to the conflict with the first transaction of I1. As I1 issues another transaction, the delay for I2 is adapted with another call to wait to consider the second conflict. In [29], a similar approach is presented that handles conflicts at the transaction boundaries. It additionally combines several atomic transactions into block transactions. If these block transactions get preempted, the transactions are split to assure that the order of data accesses is preserved.

19.4.3 TD with Conflict Handling at Quantum Boundaries

With the Quantum Giver [40], each initiator can issue multiple transactions until its individual local quantum is exceeded during the so-called simulation phase. All transactions in one quantum are executed instantaneously but use time stamping to record their start times. After the quantum is reached, the initiator informs a central timing manager, the so-called Quantum Giver, and waits for an end event. During the scheduling phase, the Quantum Giver retroactively orders all transactions according to their time stamps. It computes the delays due resource

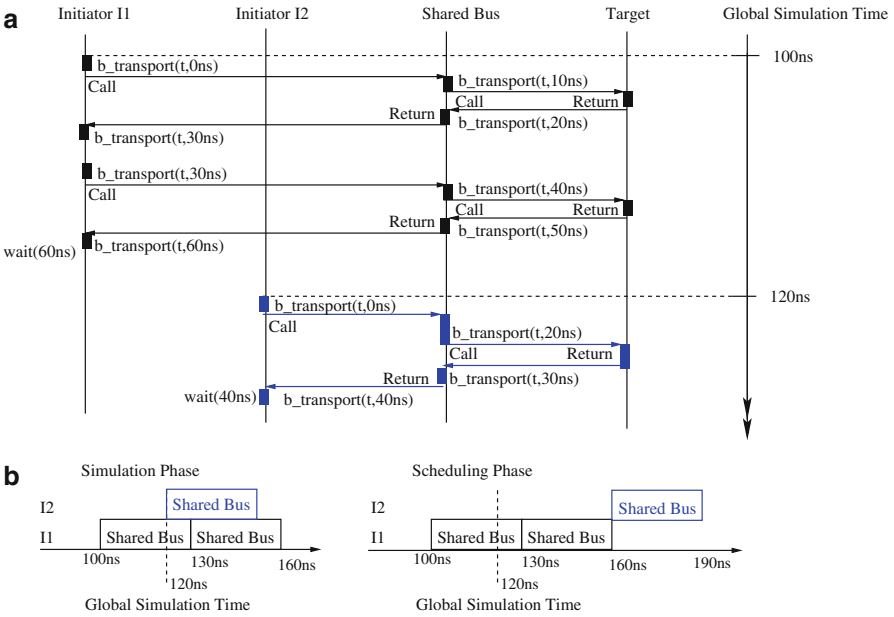


Fig. 19.12 Handling conflicts with Quantum Giver

conflicts and resolves all dependencies. According to the conflicts, the end event of each initiator is notified at the correct time. Finally, the quantum of each initiator is adjusted for the next simulation phase. The concept is illustrated in Fig. 19.12. During the simulation phase, transaction on the shared bus still overlap. The resulting delays are computed in the scheduling phase by traversing the list of transaction ordered by their starting time. The method also considers that conflicts on different shared resources might have impact on each other. This method targets fast simulation with temporal decoupling. Only a single context switch is required in each quantum, which may include several transactions. Yet, the transactions are executed immediately; thus, out-of-order accesses to shared variables must be avoided with additional synchronization guards. In [13], Advanced Temporal Decoupling (ATD) is presented. It applies TD but is a conservative approach that preserves access order. The initiators may advance their local time until they meet an inbound data dependency, e.g., a read on a shared variable. All write transactions performed on shared data are buffered by an additional communication layer. After all initiators have completed execution, the transactions are ordered, and the write transactions are completed according to their start time together with pending read transactions. This preserves the correct access order. So-called Temporal Decoupled Semaphores handle resource conflicts and compute arbitration delays. The ATD communication model is implemented in a transparent TLM (TTLM) library, which hides the implementation details from the model developer.

19.4.4 Abstract TLM+ with Conflict Handling at SW Boundaries

TLM+ is a SW-centric communication model for processors, which can only be applied for host-compiled SW simulation [7]. It not only applies TD but raises the abstraction of communication. Usually a driver function does not transfer a single data item but a range of control values together with a possible block of data. Execution of a driver function involves a complete set of bus transactions from the processor. This set of bus transactions is abstracted into a single TLM+ block transaction. The HW/SW interface is adapted in the host-compiled simulation. The software could, thus, not simulate on an ISS. Conflicts at shared resources are handled by a central timing manager, the so-called resource model. In order to give good estimates on the delay due to conflicts, the resource model requires to save a communication profile of the original driver function [19]. This profile allows to extract a demand for communication resources. Usually, a driver function would not block a shared bus completely. Accesses of different cores accessing other modules would interleave, as driver functions are executed concurrently. Yet both cores may suffer from additional delays due to arbitration conflicts. Analytic demand-availability estimators inside the resource model can be used to estimate these delays [20].

The scheduling is conducted by the resource model at the transaction boundaries. These boundaries then correspond to the entry and exit to the respective software driver function. The concept is illustrated in Fig. 19.13. Initiator I1 first executes a block transaction trigged by the execution of a driver function. At a later point, I2 starts its block transaction. I1 has higher priority; therefore, I2 is scheduled to take longer as it has not the full resource availability on the shared bus. When the block transaction of I1 finishes, I2 is not subject to further high-priority traffic blocking its bus accesses. Its end time is rescheduled to an earlier end time. This is done by event re-notification, which leads to a single call to wait() for each block transaction.

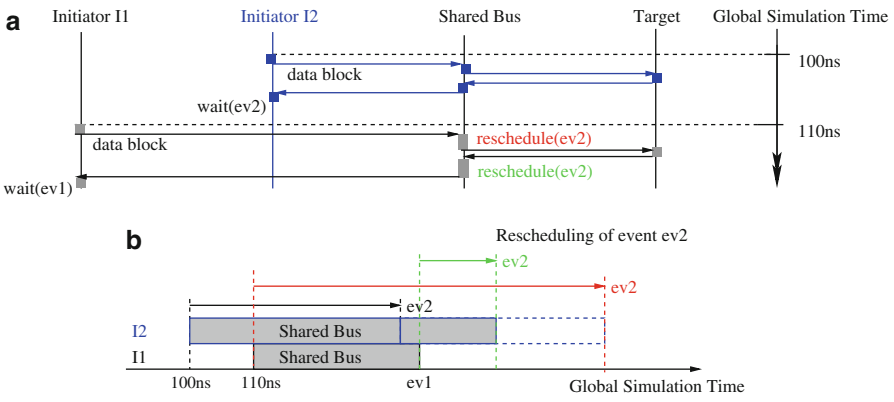


Fig. 19.13 Handling conflicts in TLM+

TLM+ targets very high abstraction and faster simulation compared to the other TLM communication methods. Yet, it does not execute the original SW because the driver functions are replaced by abstract TLM+ counterparts.

19.5 Summary and Conclusions

With time to market shrinking day by day, developing fast and accurate models is no more a luxury or good-to-have-methodology. It is essential for companies to invest in making software models for meeting their time to market. However, the fastest model is not a good model if it does not accurately match or predict the final design reality. Therefore, new methods are required that enable efficient but accurate simulation of HW/SW systems. Next-generation virtual prototypes based on host-compiled software simulation can provide such ultra-fast yet highly accurate modeling solutions.

Acknowledgments The authors acknowledge Oliver Bringmann, Wolfgang Müller, and Zhuoran Zhao for their contributions in Sects. 19.2 and 19.3.

References

1. 1666–2011 – IEEE standard for standard SystemC language reference manual (2012)
2. Bouchhima A, Bacivarov I, Youssef W, Bonaciu M, Jerraya A (2005) Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)
3. Bouchhima A, Gerin P, Petrot F (2009) Automatic instrumentation of embedded software for high level hardware/software co-simulation. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)
4. Cai L, Gerstlauer A, Gajski D (2004) Retargetable profiling for rapid, early system-level design space exploration. In: Proceedings of the 41st annual conference on design automation. ACM, San Diego, pp 281–286. doi:10.1145/996566.996651. http://portal.acm.org/citation.cfm?id=996566.996651
5. Chakravarty S, Zhao Z, Gerstlauer A (2013) Automated, retargetable back-annotation for host compiled performance and power modeling. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS)
6. Chiou D, Sunwoo D, Kim J, Patil NA, Reinhart W, Johnson DE, Keefe J, Angepat H (2007) FPGA-accelerated simulation technologies (FAST): fast, full-system, cycle-accurate simulators. In: Proceedings of the international symposium on microarchitecture (MICRO)
7. Ecker W, Esen V, Schwencker R, Steininger T, Velten M (2010) TLM+ modeling of embedded hw/sw systems. In: Design, automation test in Europe conference exhibition (DATE), pp 75–80
8. Faravelon A, Fournel N, Petrot F (2015) Fast and accurate branch predictor simulation. In: Proceedings of the design automation and test in Europe conference. ACM, pp 317–320
9. Gandhi D, Gerstlauer A, John L (2014) FastSpot: host-compiled thermal estimation for early design space exploration. In: Proceedings of the international symposium on quality electronic design (ISQED)
10. Gerin P, Shen H, Chureau A, Bouchhima A, Jerraya A (2007) Flexible and executable hardware/software interface modeling for multiprocessor SoC design using SystemC. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)

11. Gerstlauer A, Yu H, Gajski D (2003) RTOS modeling for system level design. In: Proceedings of the design, automation and test in Europe (DATE) conference
12. He Z, Mok A, Peng C (2005) Timed RTOS modeling for embedded system design. In: Proceedings of the real time and embedded technology and applications symposium (RTAS)
13. Hufnagel S (2014) Towards the efficient creation of accurate and high-performance virtual prototypes. Ph.D. thesis. https://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/3892
14. Hwang Y, Abdi S, Gajski D (2008) Cycle-approximate retargetable performance estimation at the transaction level. In: Proceedings of the design, automation and test in Europe (DATE) conference
15. Kempf T, Dorper M, Leupers R, Ascheid G, Meyr H, Kogel T, Vanthournout B (2005) A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms. In: Proceedings of the design, automation and test in Europe (DATE) conference
16. Le Moigne R, Pasquier O, Calvez JP (2004) A generic RTOS model for real-time systems simulation with SystemC. In: Proceedings of the design, automation and test in Europe (DATE) conference
17. Lee CM, Chen CK, Tsay RS (2013) A basic-block power annotation approach for fast and accurate embedded software power estimation. In: Proceedings of the international conference on very large scale integration (VLSI-SoC)
18. Lin KL, Lo CK, Tsay RS (2010) Source-level timing annotation for fast and accurate TLM computation model generation. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)
19. Lu K, Muller-Gritschneder D, Schlichtmann U (2012) Accurately timed transaction level models for virtual prototyping at high abstraction level. In: Design, automation test in Europe conference exhibition (DATE), pp 135–140
20. Lu K, Muller-Gritschneder D, Schlichtmann U (2013) Analytical timing estimation for temporally decoupled tlms considering resource conflicts. In: Design, automation test in Europe conference exhibition (DATE), pp 1161–1166
21. Lu K, Muller-Gritschneder D, Schlichtmann U (2013) Memory access reconstruction based on memory allocation mechanism for source-level simulation of embedded software. In: Proceedings of the Asia and South Pacific design automation conference (ASP-DAC)
22. Meyerowitz T, Sangiovanni-Vincentelli A, Sauermann M, Langen D (2008) Source-level timing annotation and simulation for a heterogeneous multiprocessor. In: Proceedings of the design, automation and test in Europe (DATE) conference
23. Miramond B, Huck E, Verdier F, Benkhelifa MEA, Granado B, Aichouch M, Prevotet JC, Chillet D, Pillement S, Lefebvre T, Oliva Y (2009) OveRSoC: a framework for the exploration of RTOS for RSoC platforms. Int J Reconfig Comput 2009(450607):1–18
24. Mueller-Gritschneder D, Lu K, Schlichtmann U (2011) Control-flow-driven source level timing annotation for embedded software models on transaction level. In: EUROMICRO conference on digital system design (DSD)
25. Pedram A, Craven D, Gerstlauer A (2009) Modeling cache effects at the transaction level. In: Proceedings of the international embedded systems symposium (IESS)
26. Plyaskin R, Wild T, Herkersdorf A (2012) System-level software performance simulation considering out-of-order processor execution. In: 2012 international symposium on system on chip (SoC)
27. Posadas H, Díaz L, Villar E (2011) Fast data-cache modeling for native co-simulation. In: Proceeding of the Asia and South Pacific design automation conference (ASPDAC)
28. Posadas H, Damez JA, Villar E, Blasco F, Escuder F (2005) RTOS modeling in SystemC for real-time embedded SW simulation: a POSIX model. Des Autom Embed Syst 10(4):209–227
29. Radetzki M, Khaligh R (2008) Accuracy-adaptive simulation of transaction level models. In: Design, automation and test in Europe, DATE'08, pp 788–791
30. Razaghi P (2014) Dynamic time management for improved accuracy and speed in host-compiled multi-core platform models. Ph.D. thesis, The University of Texas at Austin

31. Razaghi P, Gerstlauer A (2011) Host-compiled multicore RTOS simulator for embedded real-time software development. In: Proceedings of the design, automation test in Europe (DATE) conference
32. Razaghi P, Gerstlauer A (2012) Automatic timing granularity adjustment for host-compiled software simulation. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC)
33. Razaghi P, Gerstlauer A (2012) Predictive OS modeling for host-compiled simulation of periodic real-time task sets. IEEE Embed Syst Lett (ESL) 4(1):5–8
34. Razaghi P, Gerstlauer A (2013) Multi-core cache hierarchy modeling for host-compiled performance simulation. In: Proceedings of the electronic system level synthesis conference (ESLsyn)
35. Razaghi P, Gerstlauer A (2014) Host-compiled multi-core system simulation for early real-time performance evaluation. ACM Trans Embed Comput Syst (TECS) 13(5s). http://dl.acm.org/citation.cfm?id=2660459.2678020
36. Schirner G, Dömer R (2007) Result oriented modeling a novel technique for fast and accurate TLM. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 26(9):1688–1699
37. Schirner G, Dömer R (2008) Introducing preemptive scheduling in abstract RTOS models using result oriented modeling. In: Proceedings of the design, automation and test in Europe (DATE) conference
38. Schirner G, Gerstlauer A, Dömer R (2010) Fast and accurate processor models for efficient MPSoC design. ACM Trans Des Autom Electron Syst (TODAES) 15(2):10:1–10:26
39. Stattelmann S, Bringmann O, Rosenstiel W (2011) Dominator homomorphism based code matching for source-level simulation of embedded software. In: Proceedings of the seventh IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis
40. Stattelmann S, Bringmann O, Rosenstiel W (2011) Fast and accurate resource conflict simulation for performance analysis of multi-core systems. In: Design, automation test in Europe conference exhibition (DATE), 2011
41. Stattelmann S, Bringmann O, Rosenstiel W (2011) Fast and accurate source-level simulation of software timing considering complex code optimizations. In: 2011 48th ACM/EDAC/IEEE design automation conference (DAC)
42. Stattelmann S, Gebhard G, Cullmann C, Bringmann O, Rosenstiel W (2012) Hybrid source-level simulation of data caches using abstract cache models. In: Proceedings of the design, automation test in Europe (DATE) conference
43. Wang Z, Henkel J (2012) Accurate source-level simulation of embedded software with respect to compiler optimizations. In: Proceedings of the design, automation test in Europe (DATE) conference
44. Wang Z, Henkel J (2013) Fast and accurate cache modeling in source-level simulation of embedded software. In: Design, automation test in Europe conference exhibition (DATE), pp 587–592. doi:10.7873/DATE.2013.129
45. Wang Z, Herkersdorf A (2009) An efficient approach for system-level timing simulation of compiler-optimized embedded software. In: Proceedings of the design automation conference (DAC)
46. Zabel H, Müller W, Gerstlauer A (2009) Accurate RTOS modeling and analysis with SystemC. In: Ecker W, Müller W, Dömer R (eds) Hardware-dependent software: principles and practice. Springer, Berlin
47. Zhao Z, Gerstlauer A, John LK (2017) Source-level performance, energy, reliability, power and thermal (PERPT) simulation. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 36(2):299–312

Precise Software Timing Simulation Considering Execution Contexts

20

Oliver Bringmann, Sebastian Ottlik, and Alexander Viehl

Abstract

Context-sensitive software timing simulation enables a precise approximation of software timing at a high simulation speed. The number of cycles required to execute a sequence of instructions depends on the state of the microarchitecture prior to the execution of that sequence, which in turn heavily depends on the preceding instructions. This is exploited in context-sensitive timing simulation by selecting one of multiple pre-calculated cycle counts for an instruction sequence based on the control flow leading to a particular execution of the sequence. In this chapter, we give an overview of this concept and present our context-sensitive simulation framework. Experimental results demonstrate that our framework enables an accurate and fast timing simulation for software executing on current commercial embedded processors with complex high-performance microarchitectures without any slow, explicit modeling of components such as caches during simulation.

Acronyms

BB	Basic Block
BLS	Binary-Level Simulation
CFG	Control-Flow Graph
DSE	Design Space Exploration
FPGA	Field-Programmable Gate Array
ICFG	Interprocedural Control-Flow Graph

O. Bringmann (✉)
Wilhelm-Schickard-Institut, University of Tübingen, Tübingen, Germany

Embedded Systems, University of Tübingen, Tübingen, Germany
e-mail: oliver.bringmann@uni-tuebingen.de

S. Ottlik • A. Viehl
Microelectronic System Design, FZI Research Center for Information Technology, Karlsruhe, Germany
e-mail: ottlik@fzi.de; viehl@fzi.de

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_21

MIPS	Million Instructions Per Second
PSTC	Path Segment Timing Characterization
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data
SLS	Source-Level Simulation
TDB	Timing Database
VIVU	Virtual Inlining and Virtual Unrolling
WCET	Worst-Case Execution Time

Contents

20.1 Introduction

Hardware/software cosimulation is an essential tool during the codesign process. In principle, well-known techniques such as Register Transfer Level (RTL) simulation could be used to create simulations that provide a level of detail (e.g., cycle-by-cycle exact timing) that is sufficient for most purposes. However, the low simulation performance makes them impractical in many scenarios while this level of detail is unnecessary for many use cases. For example, a developer analyzing software performance is likely not directly interested in the contents of the branch history table of the processor, but only indirectly in its influence on the time required to execute a particular function. A simulation used by this developer does not need to simulate the branch history table if it can still accurately approximate its influence on performance. Therefore, simulation performance can be improved by raising the abstraction level, if a reasonable accuracy can be maintained.

In this chapter, we discuss an approach that is capable of accurately approximating the timing influence of the full microarchitecture, yet at the same time allows

for a highly efficient simulation as no components of the microarchitecture are explicitly simulated. This is achieved by preestimating target code instruction block timings in a Path Segment Timing Characterization (PSTC). For each block, timings are differentiated by the execution paths and thus the instructions executed before a particular execution of the block. So-called contexts serve as an abstraction of these paths. Context essentially enable an approximate consideration of the different possible states of the microarchitecture before and its impact on the execution time of a particular block execution. We implemented this approach in a highly flexible framework that support both static and dynamic methods for the PSTC. Results are stored in a common format, the so-called Timing Database (TDB). We integrated our timing simulation with QEMU [3], a fast functional simulator for binary code, which enables us to support a wide range of commercially embedded processors.

This chapter is organized as follows: In Sect. 20.2 we introduce fundamental concepts of context-sensitive timing simulation and give an overview of the current state of the art. In Sect. 20.3 we present our simulation framework and detail the main aspects of the context-sensitive timing simulation. Experimental results are presented in Sect. 20.4. A discussion of our main findings is provided in Sect. 20.5. This chapter is concluded in Sect. 20.6.

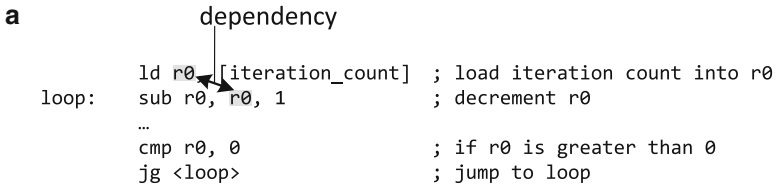
20.2 Context-Sensitive Simulation Fundamentals

In this section we first introduce the basic idea of context-sensitive timing simulation in Sect. 20.2.1. The necessary background on control-flow graphs and context mappings is introduced in Sects. 20.2.2 and 20.2.3. Related work is summarized in Sect. 20.2.4. We discuss various challenges that must be handled by context-sensitive simulation approaches in Sect. 20.2.5.

20.2.1 Basic Idea of Context-Sensitive Simulation

Binary-Level Simulation (BLS) and Source-Level Simulation (SLS), which are, respectively, discussed in ▸ Chaps. 18, "Multiprocessor System-on-Chip Prototyping Using Dynamic Binary Translation" and ▸ 19, "Host-Compiled Simulation", are well-known techniques to simulate software functionality in hardware/software cosimulation at a high performance. However, providing accurate approximations of time at this abstraction level is challenging. Typically software timing simulations consider the processor pipeline, branch prediction, and the memory subsystem – caches in particular. However, a detailed simulation of all relevant components and their interactions is extremely slow. While this approach is common in interpretative instruction-set simulation (e.g., Gem5 [4]), it degrades overall simulation performance of BLS to a level that makes the functional simulation performances nearly irrelevant [6, 22]. Such a detailed timing simulation is therefore not reasonable, if BLS or the significantly faster SLS is utilized to achieve a significant boost in simulation performance.

a



b

```
ld r0, [iteration_count]
sub r0, r0, 1             ; delayed until load complete
…
cmp r0, 0
jg <loop>
sub r0, r0, 1             ; not delayed
…
cmp r0, 0
jg <loop>
```

Fig. 20.1 Example of execution order/path dependent execution time. (**a**) Target code. (**b**) Instruction sequence executed by processor

A reasonable trade-off between timing simulation accuracy and overhead over a purely functional simulation can be achieved by advancing time by a number of processor cycles for each execution of an instruction block, for example, as discussed by Tach, Tamiya, Kuwamura, and Ike [26] for BLS. Simulation performance is improved by analyzing the processing of an instruction block by the processor pipeline only once. The results of this analysis are reused on each execution of a block. Further influences of the microarchitecture (e.g., caches) are accounted for by modeling the respective components during the simulation and applying timing penalties for certain events (e.g., cache misses). Usually, these events are assumed to not occur during the pipeline analysis, and only a single timing is calculated per block.

A major drawback of these approaches is that influences of preceding blocks (e.g., instruction dependencies) on the execution time of a block are not reflected due to the use of a single value and an isolated consideration of each block. The problem is illustrated in Fig. 20.1 for a simple loop. Before the loop is entered, the iteration count is loaded into a register, which is used as loop counter. The loop counter register is decremented by the first instruction in the loop body. In the first loop iteration, this instruction is directly preceded by the load in the instruction sequence executed by the processor and can only be processed once the loaded is completed. In subsequent iteration this is not an issue. The timing of the instruction block containing the decrement, thus, depends on which block is executed beforehand: the block preceding the loop (i.e., containing the load) or the loop block (i.e., itself). This issue can significantly impair simulation accuracy and becomes even more significant for the deeper and wider pipelines found in high-performance processors.

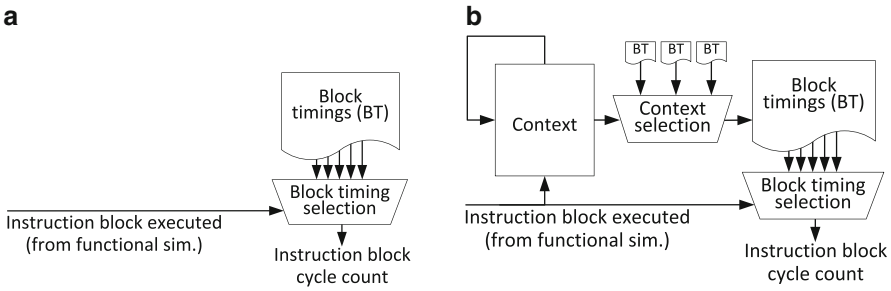


Fig. 20.2 Quintessential difference between context-insensitive and context-sensitive timing simulation. (**a**) Context-insensitive simulation. (**b**) Context-sensitive simulation

Context-sensitive simulation aims to improve simulation accuracy by enabling a consideration of the influence of preceding instructions on the timing of an instruction block. Multiple values are provided for each block, one of which is selected for a particular execution of a block based on the current context. The context essentially abstracts the precedingly executed instructions. This difference between context-sensitive and context-insensitive block-level timing simulation is illustrated in Fig. 20.2. A context-insensitive simulation selects the timing purely based on the currently executing instruction block, whereas a context-sensitive simulation can additionally consider the execution history by means of the current context.

The use of context-sensitive block timings can provide further advantages, because they can accurately reflect the influence of caches and branch prediction without an explicit model: Firstly, while online cache and branch prediction models are much faster than an online model of the full processor, they can still lead to a significant overhead [13, 26]. Secondly, a fixed penalty for cache misses and branch mis-predictions essentially corresponds to assuming the pipeline is halted in these situations, whereas, for example, in a typical super-scalar processor only instruction that depend on the result of the instruction that caused a cache miss will be stalled.

The main drawback of context-sensitive simulation is the increased complexity of calculating block cycle counts. All published approaches, therefore, require an analysis step to derive these values before an actual simulation can be performed. However, such an PSTC, that is an ahead-of-time partial characterization of timing properties of execution path segments, is also necessary for other simulation approaches (cf. ▸ Chap. 21, "Timing Models for Fast Embedded Software Performance Analysis"). For context-sensitive simulations, the PSTC needs to calculate possible execution paths through the software and characterize segments of these paths regarding their timing properties. PSTC results are then utilized to simulate the actual timing of the simulated path during the simulation. The PSTC can be static or dynamic. A static PSTC considers multiple (or even all) possible executions of the software in a single analysis. A dynamic PSTC considers only a single execution of the software at a time.

20.2.2 Control-Flow Graphs

The control flow of a program can be formally expressed by a Control-Flow Graph (CFG). A CFG is a directed graph where nodes represent Basic Blocks (BBs) and edges represent possible control flow between them. Basic blocks can only be entered at the beginning and left at the end by control-flow changes. They usually represent particular code sequences, but sometimes additional empty nodes that do not represent actual code are also included. Here, we are only concerned with CFGs of binary code without such empty nodes. However, CFGs are also used on other abstraction levels such as source code.

In a binary-level CFG, basic blocks cover sequences of processor instructions, as shown in Fig. 20.3b. Branch instructions establish the control flow and thus basic block boundaries. In a basic block, branches can only occur as the last instruction. Furthermore, instructions that can be executed after a branch (i.e., branch targets and instructions directly after a branch) can only occur as the first instruction.

For a context-sensitive timing simulation, basic knowledge of the CFG for the simulated software is required and usually reconstructed from binary code. For some approaches, a simple subdivision into blocks and edges is sufficient. However, more complex approaches operate on an Interprocedural Control-Flow Graph (ICFG), where blocks are grouped into routines, and edges carry additional semantic information to express various types of control flow (e.g., calls and returns). Additionally, loops can be represented as independent, recursive routines, and loop-back edges are represented as recursive routine calls. These concepts are illustrated in Fig. 20.3 for the source code shown in Fig. 20.3a. Typically, the function itself would be represented in a ICFG as shown in Fig. 20.3c, whereas with loop extraction a ICFG as shown in Fig. 20.3d is used. In such an ICFG, the term *routine* may also refer to extracted loops. The transformation of loop to recursive routines is only performed for so-called natural loops. In practice, this is not a significant restriction, as this condition is fulfilled for typical code of reasonable quality [14].

20.2.3 Context Mappings

The set of instruction sequences that can be executed before a particular block of instructions can be expressed by the set of ICFG paths that lead to it. In the presence of loops and recursions, this set can be infinite. Hence, it is infeasible to provide a separate timing value for each path leading to a particular block of instructions. Therefore context-sensitive timings are only provided for a finite number of subsets of the set of control-flow paths. These subsets are referred to as *contexts*. A function that maps a control-flow path to a context is referred to as a *context mapping*. This function must be chosen such that the set of contexts is finite.

This consideration and the concept of contexts is well known in the domain of static program analysis, where alternative context concepts are also known (e.g., parameter-value-based contexts). Here, we restrict our discussion to the two mappings that have been applied to timing simulation in the literature: The n-block

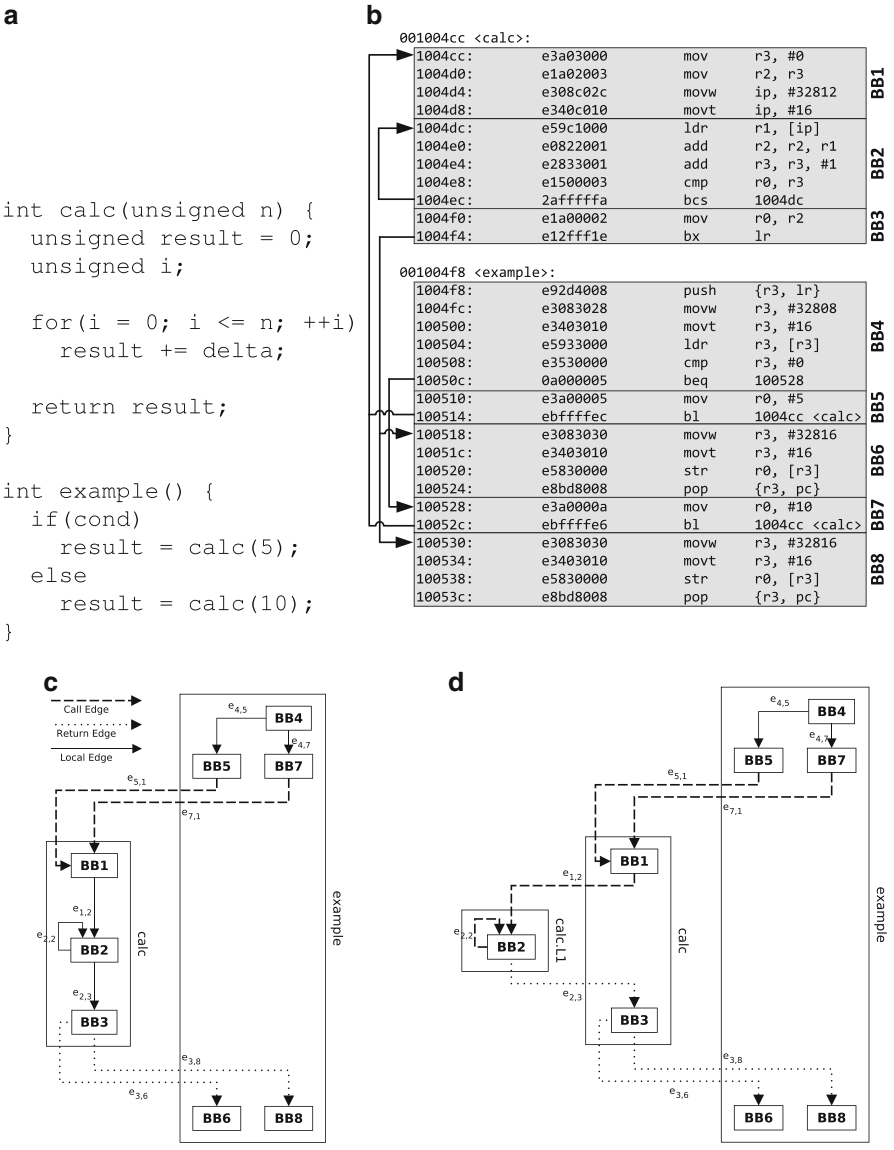


Fig. 20.3 Example of control-flow representations. (**a**) Source code. (**b**) Target code subdivided into basic blocks. (**c**) Plain ICFG. (**d**) ICFG with extracted loop

mapping and the Virtual Inlining and Virtual Unrolling (VIVU) mapping. A simplified comparison of the two mappings is shown in Fig. 20.4. The example shows a control-flow path through the example CFG from Fig. 20.3 to the second iteration of the loop. We will use this example in the following to explain both mappings.

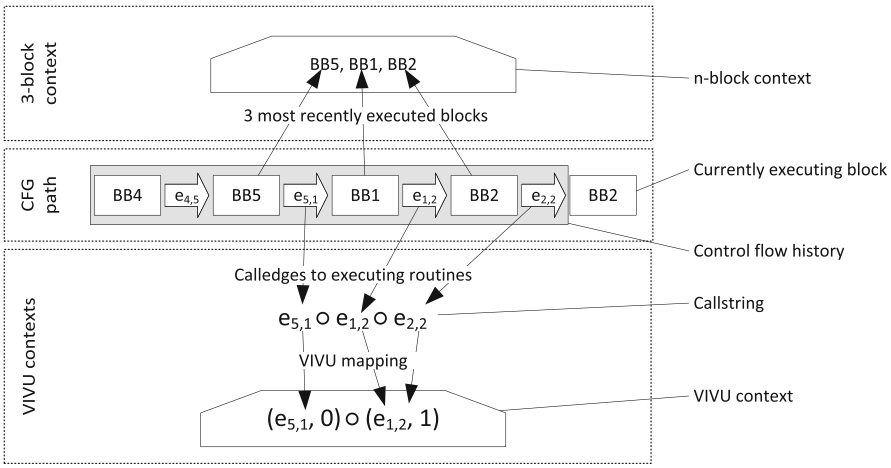


Fig. 20.4 Example of context mappings (cf. Fig. 20.3)

As VIVU only considers call edges, it only differentiates contexts by a subset of the preceding blocks. Therefore, VIVU can consider much longer histories, but at a smaller granularity. This has the advantage that the timing influence of states that are maintained much longer, such as cache contents, can be reflected accurately without explicit modeling. In practice this means that a simulation using VIVU does not necessarily require an online cache model, whereas it is necessary when using an n-block mapping. However, this advantage comes at the cost of being less sensitive to more local influences such as instruction dependencies. We would expect a combination of VIVU and an n-block mapping to provide the overall best results, but currently find such an extension unnecessary based on the very high simulation accuracy provided by the VIVU mapping in our experiments.

20.2.3.1 n-Block Mapping

For the n-block mapping, the control-flow path to the currently executing basic block is expressed as a sequence of executed blocks. In an n-block mapping, this path is simply truncated to the last n elements to form a context. In more practical terms, contexts are differentiated by the n most recently executed basic blocks.

This mapping has two main advantages: Firstly, it is relatively simple and therefore straightforward to apply. Secondly, it always includes all most recently executed instructions to differentiate contexts and can therefore provide a high accuracy for local timing influences, such as instruction dependencies.

The main drawback of this mapping is that only small values of n can be chosen, as otherwise an excessive number of contexts has to be considered. This limits the ability of this mapping to accurately reflect global timing influences such as caches. For example, the path from Fig. 20.4 would be extended by BB2 for each further loop iteration. Even for the largest published $n = 16$ [20], only the first 16 iterations of the loop could be differentiated, as afterward the context would always be a sequence of 16 instances of BB2.

20.2.3.2 VIVU Mapping

The VIVU mapping [14] is an approach to enhancing the accuracy of interprocedural program analysis with a special consideration of loops. It has been successfully applied in static Worst-Case Execution Time (WCET) analysis of binary code [9]. Our presentation of the VIVU mapping is based on the notation used by Theiling in his dissertation [27]. A sequence of a followed by b is denoted as $a \circ b$. The empty sequence is denoted as ε.

For VIVU the control-flow path to the currently executing basic block is expressed as a sequence of edges. Compared to the n-block mapping, VIVU reduces the accuracy for local timing influences, but can accurately reflect global influences. VIVU is usually applied to an ICFG, where loops are represented as recursive routines (cf. Sect. 20.2.2). Thereby, besides function calls, a call string also reflects loop iterations counts. As shown in Fig. 20.4, VIVU does not consider the whole path, but only edges that represent calls to routines that are currently executing (i.e., have not returned yet). These edges are referred to as *call edges*, and the sequence of call edges is referred to as *call string*. For example, after the loop is left, the call string would be $e_{5,1}$, as $e_{1,2}$ and $e_{2,2}$ represent calls to the loop.

VIVU maps a call string to a sequence of so-called context links. A *context link* is a pair (e, x) of a call edge e and a recursion count of x. A sequence of context links is referred to as a *VIVU context*. The call string is mapped to a VIVU context by iteratively extending the empty context ε by each call edge in the call string using the *VIVU context connector* $\oplus_n^k$ and starting with the first (i.e., oldest) call edge. The context ε is empty in terms of a context as a sequence of context links. In terms of a context as a set of control-flow paths it is the exact opposite, that is the context for all control-flow paths. The call string $e_{5,1} \circ e_{1,2} \circ e_{2,2}$ in Fig. 20.4 is mapped to $((\varepsilon \oplus_n^k e_{5,1}) \oplus_n^k e_{1,2}) \oplus_n^k e_{2,2})$. The context connector $\oplus_n^k$ applies one of two rules. It can be adapted by the parameters n and k to influence the granularity of the contexts. A particular VIVU mapping is denoted as VIVU(n,k). n is an upper limit for the recursion count, while k limits the number of context links in a context. Both k and n can be unbounded, which is, respectively, denoted as $n = \infty$ or $k = \infty$. Their exact functionality is discussed below.

Which of the two rules is applied depends on whether the connected edge (right-hand side) has the same destination as any edge in a context link in the to-be-extended context (left-hand side). An edge where this condition is true w.r.t. the to-be-extended context is referred to as recursive.

If the edge is not recursive, the context is extended by a context link with that edge and a recursion count of 0. For example, $(\varepsilon \oplus_n^k e_{5,1}) \oplus_n^k e_{1,2} = (e_{5,1}, 0) \oplus_n^k e_{1,2} = (e_{5,1}, 0) \circ (e_{1,2}, 0)$. If the length of the new context would exceed k, it is shortened to the last k elements. With $k = 1$, for example, $(e_{5,1}, 0) \oplus_n^k e_{1,2} = (e_{1,2}, 0)$. The context link $(e_{5,1}, 0)$ is dropped to meet the length restriction. For $k = 0$, all call strings are mapped to ε.

If the edge is recursive, the context is truncated to end with the context link with the same destination, and its recursion count is incremented by 1. For example, $((e_{5,1}, 0) \circ (e_{1,2}, 0)) \oplus_n^k e_{2,2} = (e_{5,1}, 0) \circ (e_{1,2}, 1)$, as $e_{1,2}$ and $e_{2,2}$ have the same destination. The recursion count is further bounded to the value of n. Theiling [27] denotes the recursion count in this case as $\top$, we do not make this distinction. For

$n = 1$, $((e_{5,1}, 0) \circ (e_{1,2}, 1)) \oplus_n^k e_{2,2} = (e_{5,1}, 0) \circ (e_{1,2}, 1)$, because the recursion count is limited to 1. As a more complex example, assume the ICFG in Fig. 20.3d contained an additional edge $e_{2,1}$, because the function `calc` calls itself recursively from within the loop. A typical context would be calculated as $((e_{5,1}, 0) \circ (e_{1,2}, 1)) \oplus_n^k e_{2,1} = (e_{5,1}, 1)$. The context link $(e_{1,2}, 1)$ is dropped because the context is truncated to end with the context link containing $e_{5,1}$, as it has the same destination as $e_{2,1}$.

In some cases the interaction between rules and parameters can become complex. For $k = 1$, $((e_{5,1}, 0) \oplus_n^1 e_{1,2}) \oplus_n^1 e_{2,1} = (e_{1,2}, 0) \oplus_n^1 e_{2,1} = (e_{2,1}, 0)$. Because $(e_{5,1}, 0)$ is dropped in the first application of the context connector, $e_{2,1}$ is not considered recursive anymore in the second application of the context connector.

To ensure a finite set of contexts when $n = \infty$, an edge-specific limit $n_{\max}(e)$ is required and applied in the same way as n. In a static PSTC, this value can be derived automatically in some cases and has to be specified by a user otherwise. In a dynamic PSTC it can formally be assumed to be equal to the number of traversals of an edge that in a particular analysis and thus ignored in practice.

20.2.4 Related Work

Here we provide a brief introduction to closely related work, a more comprehensive overview can be found in our previous publications [16, 17, 25], which also served as a foundation of this chapter and ▸ Chap. 21, "Timing Models for Fast Embedded Software Performance Analysis".

Chakravarty, Zhao, and Gerstlauer [5] presented a SLS with a 1-block mapping. A similar, but more general, concept for a BLS for an n-block mapping with $n \leq 16$ has been presented by Plyaskin, Wild, and Herkersdorf [19–21]. Stattelmann [23] applied VIVU contexts in a SLS. In our framework we also apply VIVU.

The main difference between the use of VIVU in our simulation framework and Stattelmann's simulation is that his approach only uses the mapping implicitly by essentially tracing the path through a so-called expanded supergraph [14]. The expanded supergraph is an ICFG where a node is copied for each context that node can be executed in. While this approach simplifies an efficient implementation, the inherent loss of the meaning of a context restricts the handling of contexts. Most importantly, control flow that was not considered during the preceding static PSTC is not handled in Stattelmann's simulation.

20.2.5 Challenges in Context-Sensitive Simulation

While context-sensitive simulation has many advantages to offer, the increased complexity in the required understanding of a program also poses several challenges that we discuss in the following.

20.2.5.1 Incomplete Data

At its core a context-sensitive simulation requires knowledge of the program control flow during the PSTC. It can be obtained by static or dynamic techniques. In both cases discovering the full control flow is nontrivial: Dynamic analysis techniques are

restricted to the control flow of the observed executions, which in turn depends on the used input. While static techniques do not suffer from this issue, it is complicated by computed and indirect branches (e.g., function pointers) and asynchronous changes of control flow (e.g., interrupts). In practice achieving a sufficiently high coverage is nontrivial in a single dynamic or static PSTC.

Three types of coverage issues can be differentiated:

Code — If the PSTC fails to discover parts of the code the respective nodes are missing from the reconstructed control flow graph. For example, for a program with an interrupt handler, a static PSTC will not discover the handler code, as typically no explicit control flow to the handler is contained in a program. In a dynamic PSTC on the other hand, the handler could be missing if the interrupt is not raised during the observation.

Control-Flow — A PSTC can also fail to discover edges between nodes. For example, when it fails to discover all possible call sites for a function pointer.

Context — A dynamic PSTC is unlikely to execute a program in all contexts that can occur during a simulation. However, this issue can also occur in static PSTC as a consequence of limited code or control-flow coverage.

20.2.5.2 Context Granularity

Even for small benchmarks, the large number of possible control-flow paths can lead to an excessive number of contexts. An increased number of contexts typically reduces the simulation performance, due to the increased memory footprint of the timing data and the increased complexity of context selection. On the other hand, with only few contexts, a simulation will provide a reduced accuracy.

The number of contexts is therefore a trade-off between simulation accuracy and performance. Furthermore, different contexts can be identical in terms of timing, as contexts only indirectly reflect the state of the microarchitecture. Most context mappings can be parametrized to influence the granularity of contexts. Furthermore it is possible to automatically remove unnecessary contexts, without reducing simulation accuracy.

20.2.5.3 Execution Variations

Context-sensitive timing simulations achieve a good trade-off between simulation accuracy and performance by preestimating many influences on software timing. A drawback of this approach is that variations between the simulation and the PSTC are not trivial to handle. Variations in the following areas can be considered:

Input — If dynamic PSTC is used, it is desirable that the simulation is not restricted to the executions that were observed during the analysis – in particular for changes to program input including hardware stimulations such as interrupts. This can firstly lead to the aforementioned coverage issues, but can cause subtle changes in timing behavior.

Hardware As large, or all, parts of the hardware components that influence software timing are considered before the simulation, different simulation cannot reflect different hardware configurations without a reexecution of the PSTC.

Code If the analyzed code changes, the analysis results become stale. Current context-sensitive approaches require a full reexecution of the PSTC in this case. In the future more advanced analyses may allow an adaption of the exiting timing data.

20.3 Context-Sensitive Simulation Framework

An overview of our context-sensitive timing simulation framework is shown in Fig. 20.5. As first step, context-sensitive timings of instruction blocks in a given binary code have to be obtained by a PSTC based on a detailed microarchitectural model. Our framework is very flexible in what kind of analysis technique and model can be used. Currently, we support static analysis using abstract models intended for WCET analysis by abstract interpretation and dynamic analysis using actual hardware implementations of a processor. Additionally, we plan to support further models, such as Field-Programmable Gate Array (FPGA)-based processor prototypes and detailed models in Gem5 [4], in the future. Therefore, even though our framework requires a detailed model, it is retargetable with a low effort if a model is available. Supported targets of our framework so far include the ARM7TDMI, LEON3, ARM Cortex-M3, ARM Cortex-A9, and ARM Cortex-A15 processors.

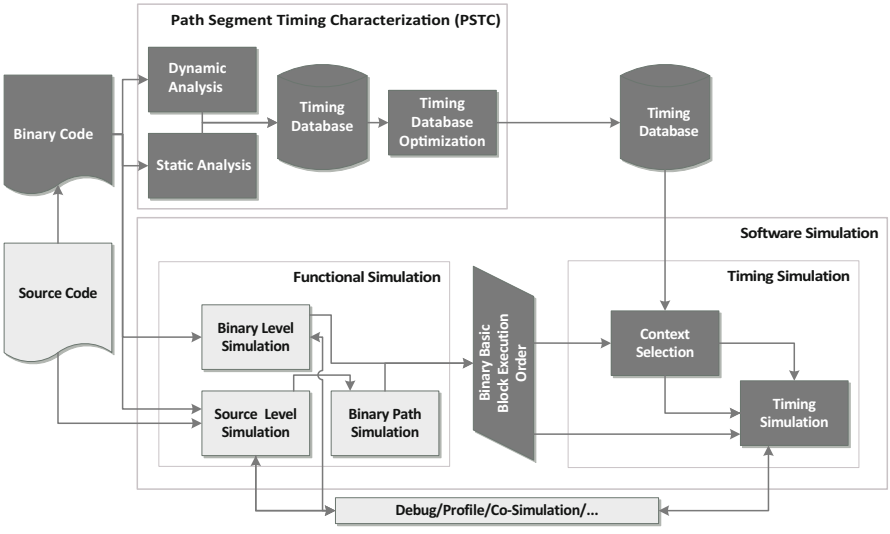


Fig. 20.5 Simulation framework overview (*grey areas*: beyond article scope)

The PSTC produces an initial TDB. The TDB includes an ICFG of the analyzed binary code and context-sensitive timing data for instruction blocks. We make use of VIVU [14] (cf. Sect. 20.2.3.2) as context mapping. This initial TDB could in principle already be used in a simulation. However, it is usually improved by applying further optimizations that lead to both, improved simulation performance and accuracy. Furthermore, it is possible to merge multiple TDBs.

After a TDB has been generated, it can be reused in multiple simulations. As a functional simulator, we support BLS based on QEMU [3] and are currently working toward the integration of an experimental SLS tool. Both produce a trace of the binary basic blocks executed in the simulation to drive the timing simulation.

Based on the basic block execution order, contexts are selected from the TDB. The current context is used to choose a timing value for a simulated block execution by which to advance simulation time. The combined functional and timing simulation for the software is then usually combined with a hardware simulation in order to provide a cosimulation. However, other use cases such as a direct profiling or debugging of the simulated software are also supported. Currently, the framework only targets single core processors. Multi- and many-core systems can be modeled in a cosimulation using multiple instances of the software simulation. A feedback between the timing and functional simulation is also possible. For example, when interrupts influence the control flow in the functional simulation and are triggered by a simulated peripheral based on the simulated timing.

20.3.1 Timing Database Contents

The TDB consists of two main parts: an ICFG and context-dependent cycle counts for instruction blocks.

20.3.1.1 Control Flow

In the TDB the control flow of the target binary code is expressed by an ICFG with loops extracted, as outlined in Sect. 20.2.2 and shown in Fig. 20.3d. To support a tracking of the current context during the simulation, edges can be marked as a return or call edge. While a call edge always represents a single call, a return edge can represent multiple returns. For example, if the program shown in Fig. 20.3 contained a return within the loop, the corresponding edges from BB2 to BB6/BB8 would represent two returns. In our current implementation, the number of returns always corresponds to the number of routines left by the return, including all directly recursive calls of these routines. More details on the handling of returns are given in Sect. 20.3.3.1.

20.3.1.2 Context-Sensitive Timings

Context-sensitive timings are differentiated by the outgoing edges of a block and an associated context. More formally, the stored timings can be thought of as a function $\text{timing}(e, c)$ that returns the timing of block b_x, when it is left via an edge $e = (b_x, b_y)$ to execute b_y in context c. An example is shown in Table 20.1 and explained in Sect. 20.3.1.3.

Table 20.1 Example of context-sensitive timings for a loop (cf. Fig. 20.3)

Context	Timings		Valid for call string	
	$e_{2,2}$	$e_{2,3}$	$e_{5,1} \circ e_{1,2}$	$e_{7,1} \circ e_{1,2} \circ e_{2,2}$
ε	5	7	Yes	Yes
$(e_{1,2}, 1)$	3	5	No	Yes, highest precedence
$(e_{1,2}, 0)$	6	6	Yes	Yes
$(e_{7,1}, 0) \circ (e_{1,2}, 0)$	5	6	No	Yes
$(e_{5,1}, 0) \circ (e_{1,2}, 0)$	3	8	Yes, highest precedence	No

In classic use of VIVU contexts for static WCET analysis, the set of known control-flow paths from an analysis starting point (e.g., the `main` function) is partitioned into contexts. Essentially, there is exactly one corresponding context for each path, which can be calculated by applying a particular VIVU mapping. However, this approach is not practical for our framework, as it should provide a best-effort approximation for paths that were not considered during TDB generation due to the limitations of the used analysis technique (cf. Sect. 20.2.5).

For example, a TDB generated from timings observed in an execution of the program in Fig. 20.3 where `calc` is called via BB5, there may not be a context for calls via BB7. This would prevent the simulation of an execution where `calc` is called via BB7 instead of BB5. In this simple example, a reasonable alternative would be a context that does not differentiate between the two call edges to `calc`. In the general case, a flexible relationship between paths and contexts must be considered to achieve the goals of our framework.

Arbitrary contexts may be combined in the TDB, and a context is considered valid for a control-flow path, if the path could be mapped to the context for arbitrary VIVU parameters n and k and an arbitrary analysis starting point. In other words, a VIVU context in the TDB can be thought of as a condition that must be fulfilled by an arbitrary suffix of the simulated path to make the respective context-sensitive timings a valid choice. The drawback of this approach is that it becomes necessary to define which of the valid contexts should be selected during simulation.

20.3.1.3 Context Precedence

As outlined in the preceding section, a TDB may – and typically does – contain multiple contexts that are valid for a path or more specifically the respective call string. One of these contexts has to be selected during the simulation such that the simulated timing is as accurate as possible. In our framework this process is based on two assumptions: First, in terms of the set of control-flow paths represented by a context, a context that represents a smaller set is more likely to be accurate because variations in the timing of fewer paths are less likely. Second, a context that represents paths that share recent control flow is likely more accurate, as it is more likely that recent control flow (i.e., more recently executed instructions) still have an impact on the timing of the currently executing block. For example, if two precedingly executed blocks caused memory to be loaded into the cache, the

memory that was loaded by the more recently executed block is more likely to still reside in the cache.

An example of context-sensitive timings and the precedence of various contexts is shown in Table 20.1 for the loop from the example program in Fig. 20.3. As can be seen each context provides timings for the outgoing edges of the blocks in the routine (BB2). As every call string may be mapped to ε, it is a valid choice for every call string, but also takes the lowest precedence. Thus it can serve as a fallback if no other context is available. For the call string $e_{5,1} \circ e_{1,2}$, the context $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ is the preferred choice, as it is more specific than $(e_{1,2}, 0)$ or ε. For the call string $e_{7,1} \circ e_{1,2} \circ e_{2,2}$, the preferred context is $(e_{1,2}, 1)$. While $(e_{7,1}, 0) \circ (e_{1,2}, 0)$ can be considered equal in terms of the first assumption, it represents less recent control flow and thus has a lower precedence. Roughly it can be said that the choice between $(e_{7,1}, 0) \circ (e_{1,2}, 0)$ and $(e_{1,2}, 1)$ is a choice between a context that reflects $e_{7,1}$ and one that reflects $e_{2,2}$. Since $e_{2,2}$ ultimately represents more recently executed instructions, $(e_{1,2}, 1)$ is the preferred context.

Since a context may represent an infinite number of paths, some of which may not have been discovered during TDB generation, using these assumptions directly to establish context precedence is not a good solution. Furthermore the efficiency of the context lookup has also been considered. The context selection scheme of the simulation framework is outlined in Sect. 20.3.3.3.

20.3.2 Timing Database Generation

The TDB can be generated using either a static or a dynamic PSTC of the application binary code for the target platform. Compared to the static PSTC, the main advantage of the dynamic PSTC-based generation is that it can provide highly accurate results even for complex system architectures. However, this requires appropriate program stimuli during the dynamic PSTC, which is not necessary for the static PSTC.

20.3.2.1 Dynamic PSTC

The dynamic PSTC-based TDB generation requires a timed trace of a program execution as an input. The trace must allow a full reconstruction of the program control flow (i.e., the exact sequence of executed instructions), whereas timing data is only required for every execution of a basic block. In practice, such traces can be obtained from on-chip tracing units available for many embedded processors. For example, they are supported by ARM CoreSight [2] and the Nexus 5001 standard [15]. Note that such tracing units typically only provide time stamps for executed branch instructions. More specifically, ARM style traces provide a time stamp for take and non-taken branches, whereas Nexus style traces only provide time stamps for taken branches. Therefore, some time stamps may cover multiple basic blocks. In this case the timing of individual block executions is interpolated.

To construct a TDB, two passes are performed over a trace. During the first pass, an ICFG is extracted. The extracted graph is analyzed to identify loops and

subsequently transformed to the form described in Sect. 20.2.2. During the second pass, the graph is traversed based on the trace, and the observed timings are stored in the TDB for the current context.

If the VIVU mapping is limited (i.e., $n \neq \infty$ or $k \neq \infty$) or the traced execution contains indirect recursions, it is possible that multiple observations are stored in the same context. In this case an average is stored in the TDB. Additionally, we store the number of observations underlying a timing value as this information is required in the context generalization optimization. This optimization is applied after the TDB generation to enable an accurate simulation if the simulated control flow differs from the traced control flow. A description is given in Sect. 20.3.2.3.

20.3.2.2 Static PSTC

For the static PSTC, we adopt well-known techniques from the domain of static WCET analysis. More specifically, we employ a static control-flow analysis to extract the ICFG, on which block timings are estimated using abstract interpretation [7]. In our current implementation, these steps are performed by Absint aiT [1], which is a commercial tool for WCET analysis, from which we read intermediate results. The intermediate results are translated to a TDB, and we typically execute multiple analyses and merge the resulting TDBs, as a single analysis may not provide sufficient coverage. This is, for example, the case for typical programs with interrupt handlers: As the analysis only discovers explicit control-flow changes, an analysis of the main function does not cover interrupt handlers and vice versa. The TDB generator could, for example, execute two distinct analyses in aiT, one starting at the main function and another at the interrupt handler, and merge their results in a single TDB.

As aforementioned, the control-flow representation and context mapping used in the TDB are based on research into WCET analysis. More specifically, they are based on their use in aiT and earlier work [23, 24] by our groups that essentially directly made use of the same intermediate results. However, for the TDB we make two small, but significant changes: First, aiT only considers discovered control-flow paths for each context, whereas the TDB assumes a context is valid for any control-flow path that can be mapped to it. As a result, when simulating control flow beyond the control flow considered during the analysis, the worst-case timing guarantees made by the analysis cannot be maintained. We expect that changing the static analysis to enable a generic worst-case timing simulation is feasible, but this topic is beyond the scope of our current research activities. Second, there is a difference between the ICFGs used by the static analysis and the ICFGs used in simulation that we describe in the following.

The ICFGs used during the static analysis include special empty nodes to avoid the complexity of handling various special cases. For example, calls are represented by a call and a return node that follow/precede the basic block a call is made from/returns to. As these nodes contain no instructions, their execution cannot be registered during a simulation, and we therefore do not include them in the TDB. During the translation of analysis results to a TDB, we map path between nonempty blocks that contain only empty blocks to a single edge. A resulting edge can therefore cross call and/or multiple returns. This translation is not sensible for

arbitrary graphs, but in our experience the analysis only produces graphs where this translation can be handled with a reasonable effort.

20.3.2.3 Optimization

Various optimizations can be applied to the TDB that improve simulation performance and/or accuracy. Performance improvements can be achieved by removing unnecessary contexts. Thereby the overall memory footprint of a simulation is reduced and the context lookup algorithm used in the simulation, which we describe later in Sect. 20.3.3.3, can terminate early. Accuracy improvements on the other hand are achieved by synthesizing new contexts and/or adding new data to existing contexts. These new contexts may then be used to simulate the timing of paths that were not considered during the PSTC and thus may not be represented by the context produced by the PSTC. This step is especially critical for the dynamic PSTC, as only a single execution is considered. In practice it enables a simulation of the timing of a program for multiple different inputs (and thus potentially different paths) based on the timing observed for one input. Typically, the resulting TDB is smaller after all optimizations have been applied. All optimizations are performed on a per-routine basis. They are constructed such that timing data remains unchanged for the executions considered during TDB generation.

Removing Unnecessary Contexts

Contexts can be removed from a TDB under some conditions. For example, assume a TDB contained only the contexts $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ and $(e_{5,1}, 0) \circ (e_{1,2}, 1)$ for the loop from the example in Fig. 20.3. As the contexts are not differentiated by the context link $(e_{5,1}, 0)$, the contexts $(e_{1,2}, 0)$ and $(e_{1,2}, 1)$ can be used, respectively. This is possible because these shorter contexts will be selected for each path the original contexts would be selected for, and as no other contexts exists, these were the only paths considered during analysis. Furthermore, if the context-sensitive data for both contexts is identical, $(e_{1,2}, 1)$ can be removed based on the same rationale. As $(e_{1,2}, 0)$ is the only remaining context, it can simply be replaced by ε.

In general, these optimizations are expressed by two rules that define which contexts can be considered for removal without considering the associated data and separate consideration if the data of two contexts is mergeable. The data for two contexts is considered mergeable, if cycle counts are identical for all edges that are present in both sets of data. In more practical terms, two contexts are mergeable if cycle count lookup that was successful in either context before the merge returns the same result after the merge. Note that a context without any associated data may be merged with any other context. The first rule is referred to as shorten call strings. It defines that a context $(e_1, r_1) \circ (e_2, r_2) \circ \cdots \circ (e_n, r_n)$ may be removed and merged with $(e_2, r_2) \circ \cdots \circ (e_n, r_n)$, if no other context $(e_x, r_x) \circ (e_2, r_2) \circ \cdots \circ (e_n, r_n)$ with $e_x \neq e_1 \wedge r_x \neq e_1$ exists. The second rule is referred to as merge leaf iterations. It defines that a context $(e_x, r_x) \circ \ldots$ may be removed and merged with $(e_x, r_y) \circ \ldots$, if $r_y < r_x$ and r_y is the largest value where this holds. As can be seen from the example in the beginning of this section, each rule can create a situation where the other rule can be applied. They are therefore applied in turn until a fixed point is reached.

Context Generalization

The prime motivation for the context generalization optimization is the incompleteness of a TDB created from a single program execution using the dynamic PSTC-based TDB generation. As example assume a TDB for the example in Fig. 20.3 was generated from the timings observed for an execution where `calc` is called from BB5. Such a TDB could only contain the contexts $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ and $(e_{5,1}, 0) \circ (e_{1,2}, 1)$. If `calc` is called via BB7 in a simulation, a call string for the second iteration of the loop would be $e_{7,1} \circ e_{1,2} \circ e_{2,2}$. No context could be selected for this call string. A good alternative would be to use the data from context $(e_{5,1}, 0) \circ (e_{1,2}, 1)$, that is, use the timing of the second iteration when the call was from BB5 to simulate the timing of the second iteration when the call was from BB7. This could, for example, accurately reflect that timing under consideration that the instruction cache was already filled with the loop instructions in the first iteration.

For a general understanding of this optimization, it is helpful to consider contexts as sets of control-flow paths (i.e., the set of paths that could be mapped to the context). As a first step consider the valid contexts of a call string that was not considered during the initial TDB generation. Each of these contexts represents a set of control-flow paths that includes the paths represented by the call string. For some of these potential contexts, there exist other contexts in the TDB that represent a subset of a potential context, a condition that is always at least true for ε. For the example mentioned in the preceding paragraph, a valid context for the call string $e_{7,1} \circ e_{1,2} \circ e_{2,2}$ would be $(e_{1,2}, 1)$, of which the existing context $(e_{5,1}, 0) \circ (e_{1,2}, 1)$ is a subset. The best context to generate would be the one with the highest precedence as discussed in Sect. 20.3.1.3.

However, generating these contexts is not possible before the simulation, as not all call strings may be known, and could be very inefficient during simulation. Instead the context generalization synthesizes more general versions of the existing contexts. For each context $(e_1, r_1) \circ (e_2, r_2) \circ \cdots \circ (e_n, r_n)$, new contexts $(e_2, r_2) \circ \cdots \circ (e_n, r_n)$, $(e_3, r_3) \circ \cdots \circ (e_n, r_n)$ and so on until ε are generated. Each new context is associated with average timings of the underlying original contexts. The averages are weighted by the number of underlying observations, if a dynamic PSTC was used for the initial TDB generation.

An additional issue is also handled by this optimization: The timing for some edges may only have been observed for some of the original contexts. To ensure that all context contain a timing for all edges of a routine, missing values in original and synthesized contexts are filled in from the most specific context that contains a value for that edge and is a generalized version of the context with the missing data.

20.3.3 Simulation

To make use of the context-dependent timing, appropriate contexts have to be selected during a simulation. A naive approach to look up a context from the TDB during simulation would be to maintain the current call string during the simulation, and on each change find the valid context with the highest precedence in the TDB

by applying various VIVU(n, k) mappings. However, this approach is unlikely to achieve a high simulation performance.

To enable a robust, flexible, and efficient context lookup, under consideration of the discussion of context precedence outline in Sect. 20.3.1.3, we developed the following approach: During simulation, similarly to a call string, we maintain a so-called *intermediate string*. Contexts in the TDB are structured in a per-routine *context tree*. They can be looked up by the *dynamic context selection* algorithm.

During simulation the execution of target binary code basic blocks must be registered as an event. In BLS, we perform this by instrumenting the first instruction of every basic block. In SLS, this can be achieved by so-called path simulation code [24], which reconstructs the execution path through the target binary code based on the source code execution path (cf. ▸ Chap. 19, "Host-Compiled Simulation"). Time is advanced for each executed block by the cycle count stored in the TDB for the taken outgoing edge in the current context.

20.3.3.1 Intermediate String

An intermediate string is similar and in simple cases identical to a VIVU context for $n = \infty$ and $k = \infty$. More specifically, it is a sequence of context links (i.e., a pair of a call edge and a recursion count) with an unbounded length and unbounded recursion counters. On each traversal of a call edge, the intermediate string is updated. Usually, a new context link with a recursion count of 0 is appended to the intermediate string. If an edge is a direct recursion (i.e., has the same destination as the last context link element) and is marked as a non-returning call (e.g., it is a loop back edge), no element is appended and the recursion counter of the last element is increased instead. This process is similar to the VIVU context connector, but prevents a loss of information that may be required for a context lookup or a future update of the intermediate string. In contrast to maintaining a call string, it has the advantage that no additional memory is required for loop iterations, which are the most common case of recursive calls.

An edge can only represent a single call, as every natural loop has a distinct header (control-flow structures that could intuitively be considered two distinct loops with a shared header are formally considered a single natural loop) and, in practice, functions always have a few prologue instructions. However, an edge can represent multiple returns, for example, when a loop contains a function return. In the TDB an edge can therefore be a call edge (or not) and additionally represent an arbitrary number of returns, which shall be performed before the call. The number of returns is not the number of call edges that should be removed from a call string, as this number may be statically unknown (e.g., in the aforementioned case of a return from within a loop). Instead it is defined as the number of context links that shall be removed from the intermediate string.

20.3.3.2 VIVU Context Tree

In the TDB contexts for a routine are structured as a tree, where each node corresponds to a particular VIVU context but may be empty/unused. The children of a node represent more specific versions of contexts of their parent. The root of

the tree represents the context ε and therefore is a valid context for any control-flow path. Each step from a node to a direct child is associated with a context link, where the context of the child is given by prepending the context link to the context of the parent. The full context of a node is therefore the sequence of context links for the path from the node to the root (the path from the root to the node is the reversed context). An example for the context tree is given in the next section.

20.3.3.3 Dynamic Context Selection

The dynamic context selection algorithm is used to select a particular context for an intermediate string from a context tree. Essentially, the context tree is ordered such that during a descend into the tree, each step to a child node leads to a more specific context for the intermediate string. As the path from the root to a node is the revered context for that node, the intermediate string is also processed in reverse. Usually the child to descend into is selected based on the next context link in the reversed intermediate string until no further valid child is available. An exception are recursions over multiple call edges and returnable direct recursions, where multiple children may be a valid choice and a scoring scheme is used to select a particular context. In the following, we first describe the lookup algorithm without consideration for this special case and provide a separate description afterward.

The lookup starts at the root node, which corresponds to the context ε, which is a valid choice for any intermediate string. For descending into child nodes, the intermediate string is considered in reverse order. A child node is considered a candidate for descending, if the edge of the context link for the step to the child node and the edge for the currently considered context link of the intermediate string are identical. Often (e.g., for loops) multiple candidates with different recursion counts exist. In this case the candidate with the highest recursion count is selected, that is, less than or equal to the recursion count of the currently considered context link of the intermediate string. This process is repeated until no further candidate for descending can be found. The deepest encountered node with valid timing data is selected as current context, usually this is the last node the lookup descended to. In the worst case, when no descend from the root is possible, the context ε is selected if it contains valid timing data. In our current implementation, we ensure this is always the case by an optimization of the TDB (cf. Sect. 20.3.2.3).

An example for the context tree and dynamic context selection is shown in Fig. 20.6. Both intermediate strings are evaluated in reveres. For $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ all intermediate strings are evaluated, and the context links to the child nodes and from the intermediate string are identical. For $(e_{5,1}, 0) \circ (e_{1,2}, 2)$ on the other hand, the child node for $(e_{1,2}, 1)$ is selected, as there is no child node with a higher recursion count. Furthermore, the intermediate string is not evaluated further, as there are no child nodes to descend into.

To handle recursions over multiple call edges and returnable direct recursions, the algorithm checks whether other context links of the (unreversed) intermediate string preceding the context link has an identical destination. If this is the case, it preforms additional descends for the other elements and a recursion count that is the sum of both recursion counts plus one (to account for the increment that would have been performed by the VIVU mapping but was not performed for the intermediate

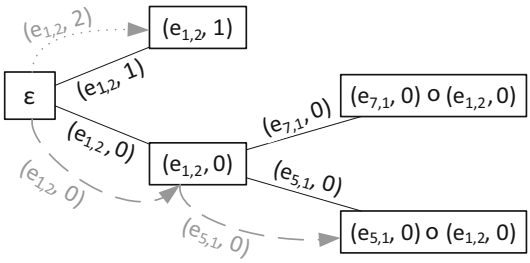


Fig. 20.6 Example of context tree (*black*) for the example contexts from Fig. 20.1 and context selection (*gray*) for intermediate strings $(e_{5,1}, 0) \circ (e_{1,2}, 0)$ (*dashed*) and $(e_{5,1}, 0) \circ (e_{1,2}, 2)$ (*dotted*)

string). The depth is not a sufficient criterion to select the most specific context in this case. Therefore, each encountered node is scored based on the sum of all recursion counts of all context links that were evaluated during the descend to this node plus the number of evaluated context links. The node with the highest score is selected and if multiple nodes have the same score the first encountered node is preferred.

20.3.3.4 Fallback Strategies

It is unlikely that a TDB provides perfect coverage of the simulated control flow for various reasons: For example, changes in control flow due to interrupts (e.g., a timer interrupt driving a preemptive scheduler) are not expressed in an ICFG. Another reason can be imperfect matchings in a SLS, which can lead to blocks missing from the path simulation code. The simulation framework includes various fallback mechanisms to handle such cases. First, the TDB provides a fallback cycle count for each block, which we currently calculate by averaging all cycle counts for the outgoing edges of a block. This value can be used if a block execution order is simulated for which no corresponding edge exists in the ICFG stored in the TDB. Second, if such an execution order is simulated the intermediate string may be incorrect, because the missing edge may have been a call or return. This is handled by removing elements from the intermediate string until the edge of the last context link is an incoming edge of the currently executing routine. In the worst case the intermediate string is cleared. For example, a program with preemptive scheduling is roughly simulated as follows: The program is simulated normally as outlined above. When control flow is diverted to the timer interrupt handler, the fallback value for the currently executing block is used to advance simulation time and the intermediate string is cleared. During the subsequent execution of the scheduler, the intermediate string is rebuild providing progressively more accurate timings. Once the interrupt handler returns to a different or the previously executing thread, a similar sequence is repeated.

20.4 Experimental Results

In this section we present experimental results for our timing simulation framework. First, we establish a baseline regarding simulation accuracy based on results for small benchmarks. Afterward, we discuss case studies for more complex

applications. Due to space restrictions, we do not provide an in-depth description of the experiments here, more details can be found in our preceding publications [16, 17].

20.4.1 Benchmarks

As benchmarks we use several programs from the Mälardalen WCET benchmark [10] collection. Each of these benchmarks focuses on specific control-flow structures, for example, typical computations such as matrix multiplications or implementation patterns such as state machines. This has the drawback that the benchmark collection does not provide representative workloads. However, it allows an evaluation of the relationship between program structure and simulation properties.

Some benchmarks were excluded for various reasons: Several benchmarks had very short execution times or were even completely optimized by the compiler. Furthermore we had to adapt benchmarks to allow a variation of inputs for our experiments using a dynamic PSTC. For these experiments we selected benchmark where such an adaptation was feasible with a reasonable effort. This adaptation was primarily necessary, as our framework can simulate the timing exactly without input variations.

We performed experiments with these benchmarks for TDB generation using static PSTC for an ARM Cortex-M3 system and using dynamic PSTC for an ARM Cortex-A9 system. The experiments using a static PSTC were evaluated regarding both simulation accuracy and performance, whereas the experiments using a dynamic PSTC were only evaluated regarding simulation accuracy.

20.4.1.1 Simulation Accuracy

Figure 20.7 shows the deviation of simulation results from hardware measurements for the Mälardalen benchmarks using four different TDBs and two different hardware configurations. The TDBs inf and opt represent the intended use of our simulation framework: a TDB that was generated using a VIVU(∞,∞) mapping in the static PSTC and, in the case of opt, further optimization of the TDB. Noctx represents a simulation that does not itself make use of contexts but relies on the results of a static PSTC that does use contexts. Whereas the zero case represents a simulation where contexts are used in neither.

Many programs can be simulated accurately by any TDB, but only the context-sensitive simulations provide accurate simulation results in all cases. The difference is more pronounced in the configuration with two wait states. Here, the increased latency of the flash the benchmarks are executed from makes the timing behavior more complex, which can only be reflected accurately by the context-sensitive simulation.

Table 20.2 shows minimum and maximum simulation errors when using a dynamic PSTC to generate a TDB. As the main issue with a dynamic analysis-based TDB generation is a variation of program inputs, we adapted a subset of the Mälardalen benchmarks to enable parameter variations. For this evaluation we chose a wide range of values for each parameter (e.g., size parameters in a range

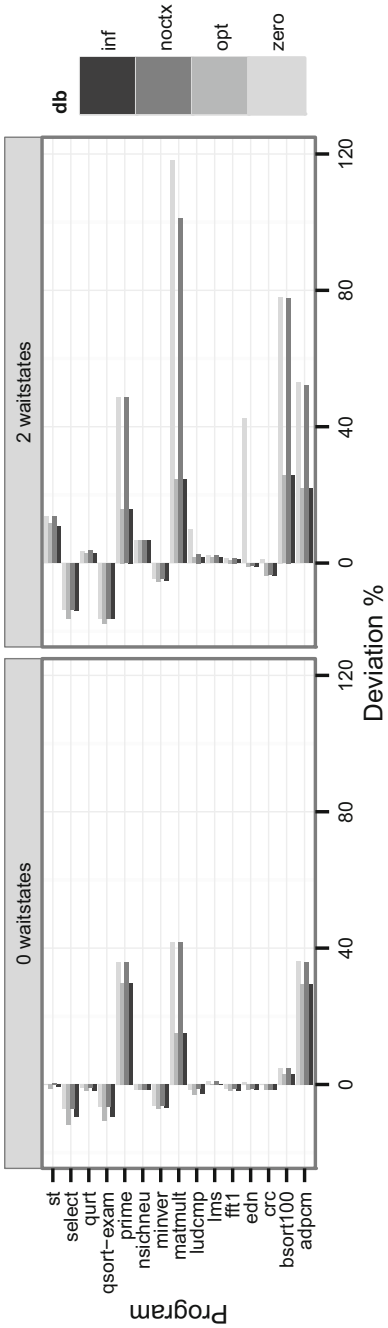


Fig. 20.7 Deviation of simulation results from hardware measurements for Cortex-M3 when using a static PSTC

Table 20.2 Simulation error when using a dynamic PSTC

Program	Param.	Avg./max. error %					
		All[a]		Single[b]		Pair[c]	
matmult	Seed	0.0	0.0	0.0	0.0	0.0	0.0
	Matrix size	12.5	28.8	8.2	16.2	4.1	12.0
crc	Input size	2.4	14.2	0.8	3.9	0.4	1.2
	Input values	0.2	0.5	0.1	0.4	0.0	0.2
adpcm	Input amp.	0.9	1.8	0.7	1.7	0.0	0.1
	Input freq.	1.0	3.1	0.2	0.6	0.1	0.4
	Num. samples	0.6	1.5	0.4	0.7	0.0	0.0
bsort100	Seed	0.0	0.0	0.0	0.0	0.0	0.0
	Array size	15.2	84.3	0.7	2.1	0.2	0.8
qsort-exam	Seed	4.2	7.0	2.8	4.0	2.8	4.0
	Array size	8.2	21.9	3.4	6.5	2.4	6.1
prime	Tested prime	11.1	29.0	6.3	12.8	2.3	5.3
select	Seed	0.7	1.8	0.5	0.9	0.3	0.6
	Array size	8.0	42.7	2.2	9.2	0.6	1.3
All	None	0.0	0.0	0.0	0.0	0.0	0.0

[a]All: All TDBs
[b]Single: on average most accurate TDB
[c]Pair: Most accurate pair of TDBs when choosing the more accurate one of the pair per program

such that the working set size exceeds the L1 cache size for some values but not for others), traced executions for each parameter value and generated a TDB from every trace. Afterward, each TDB was used to simulate executions for all parameters. We calculate mean and maximum simulation errors from these simulation for multiple scenarios of selecting a TDB for a simulation. In all we include all TDBs, which is similar to a user choosing tracing inputs arbitrarily. In single we selected the single, on average best TDB, which represents the best result a user could achieve when he deliberately selects tracing inputs. The pair selection strategy is similar, but additionally considers that a user may generate multiple TDBs – two in this case – to further improve simulation accuracy.

The control flow in most benchmarks does not change significantly when the working set size is identical in TDB generation and simulation. In these cases even using arbitrary inputs during tracing leads to a high simulation accuracy. An extreme exception to these rules is the `prime` benchmark, as it makes heavy use of a low-level math function to perform modulo operations. This function essentially contains a large switch statement, if the TDB is generated for shorter executions the switch statement is not covered sufficiently. In this case and also variations of the working set size, a more careful selection of tracing input is required. In most cases using two TDBs provides a further accuracy improvements.

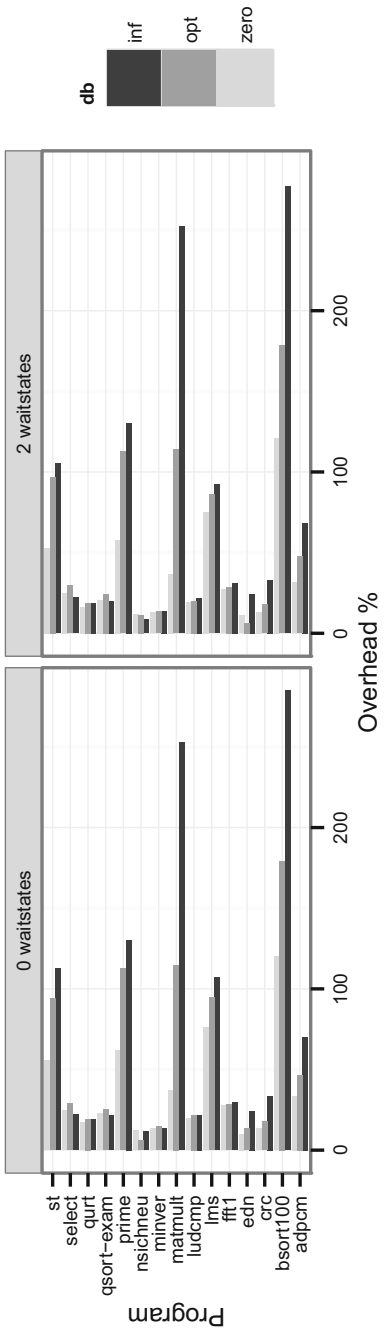


Fig. 20.8 Simulation overhead

20.4.1.2 Simulation Performance

Figure 20.8 shows the overhead of the timing simulation for the Mälardalen benchmarks in the experiments for the Cortex-M3 processor using a static PSTC. The overhead is independent of the used hardware configuration, as during the simulation the only difference between the two cases is the timing values stored in the TDB. While not shown in the figure, this has also the consequence that there is no significant difference in simulation performance for a TDB generated using a dynamic or static PSTC. A performance gain of over factor 10 can be achieved utilizing SLS and an automaton based encoding of the timing data [18].

20.4.2 Case Studies

Benchmarks are usually significantly smaller and less complex than real applications, but our simulation becomes more complex with application size and complexity. We therefore put a focus on using real code in the evaluations of our simulation framework. Here we present a few selected case studies.

20.4.2.1 eCos

Our eCos case study explores the capabilities of the simulation to exceed the limitations of a static PSTC used during TDB generation. For this purpose we ported the embedded operating system eCos [8] to our reference hardware that was also used in the experiments for the Mälardalen benchmarks using static PSTC. We configured eCos with support for preemptive multi-threading and crafted an application that executed two concurrent instances of the matmult benchmark.

The preemption creates a feedback between the temporal and the functional behavior of the program, as changes in software performance lead to additional thread switches. This factor complicates the static PSTC used to generate a TDB. In particular an accurate simulation would be impossible without dynamic context selection and the fusion of results from multiple analyses.

When using an optimized TDB, our simulation can approximate the hardware execution with an error of less than 0.25%, whereas a simulation and analysis without context lead to an error of over 70%. This large error is not only the result of the decreased accuracy of the block-level timings but also the resulting change in the functional simulation due to additional thread switches. This demonstrates the importance of an accurate low-level timing simulation in the presence of feedback between the functional and the timing behavior of an application.

While the difference in the functional behavior prevents an exact calculation of the timing simulation overhead, the difference in simulation performance is comparable to the results for the Mälardalen benchmarks: The context-sensitive simulation with an optimized TDB achieves about 22 Million Instructions Per Second (MIPS), while a simulation without timing and thus a sequential execution of both threads achieves 39 MIPS.

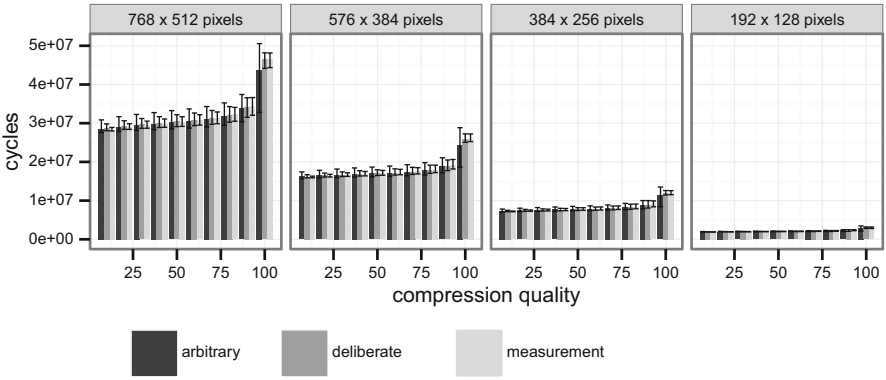


Fig. 20.9 Min/mean/max number of cycles for execution of image compression as simulated using arbitrarily and deliberately selected tracing inputs compared to hardware measurements

20.4.2.2 Image Compression

As an example for highly optimized software, we investigated the simulation accuracy for the libjpeg-turbo [12] image compression library on a Cortex-A9, which makes use of Single Instruction, Multiple Data (SIMD) instruction set extensions. We created an application that compresses images from a collection by Kodak [11] at various compression quality settings. To further vary the image size, we rescaled some of the images to obtain four groups of constant size with five images each. TDBs were generated using a dynamic PSTC and explored the relationship between tracing input and simulation accuracy.

If an arbitrary input is used in tracing, the simulation error is 2.07% on average and at most 30.63%. If more care is taken by making sure tracing and simulation image size are identical and the special case of a compression quality of 100 is handled separately, the error can be reduced to 1.18% on average and at most 11.66%. Figure 20.9 shows a comparison of simulation results with hardware measurements. As can be seen, the simulation accurately characterizes the application timing, but for the case of arbitrary tracing inputs suggests a wider variation.

20.4.2.3 Advanced Driver Assistance

As an example for a complex, modern embedded application we experimented with a video based circular traffic sign recognition on the same platform used in the image compression application. The application processes each image in two stages. First, in the segmentation stage circles are detected. In the subsequent classification stage these circles are classified as a particular traffic signs or no sign. For our experiments we varied two segmentation parameters that influence the number of circles that go into the classification stage and must be tuned in practice to achieve a reasonable trade-off between application performance and recognition accuracy. Furthermore we used several input images.

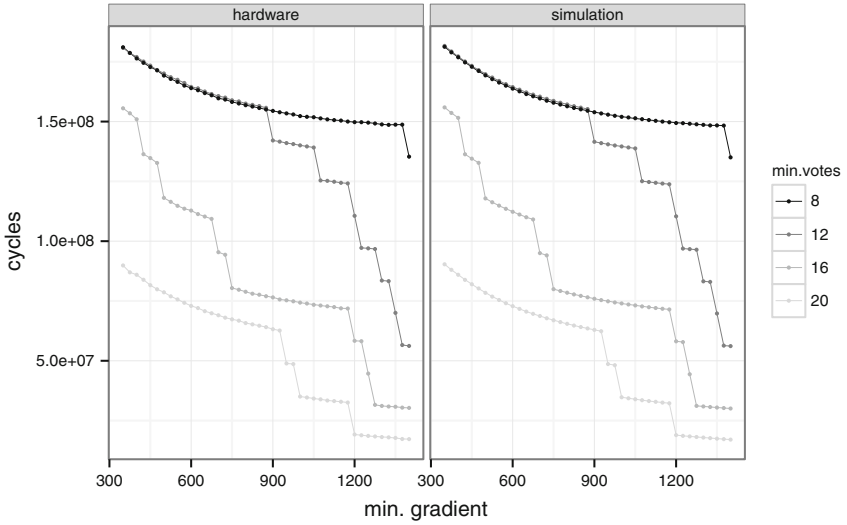


Fig. 20.10 Subset of simulation results for advanced driver assistance application compared to hardware measurements

If an arbitrary input is used in tracing, the simulation error is 0.98% on average and at most 7.55%. If more care is taken by making sure tracing and simulation images are identical or at least very similar, the error can be reduced to 0.27% on average and at most 2.06%. Figure 20.10 shows a comparison of simulation results with hardware measurements. As can be seen, the simulation slightly idealizes the application timing, which is more erratic on hardware.

20.5 Discussion

Context-sensitive software timing simulation offers many advantages over other approaches to software timing simulation. However, the need for a PSTC of the software code and the complexity of the control-flow models as well as context mappings lead to a number of limitations. In this section we list advantages and limitations of context-sensitive simulation at the current state of the art.

20.5.1 Advantages

While the simulation at its core is still driven by events at a higher level of abstraction than individual processor cycles, the use of contexts allows an accurate approximation of the flow of instructions through the processor pipeline from a timing perspective. As a consequence, context-sensitive timing simulation can offer

an extremely high accuracy, in particular, when used with a dynamic PSTC that can also consider application behavior for typical workloads.

Furthermore, if VIVU is used as context mapping, contexts can also approximate cache effects. This makes online cache models unnecessary, which otherwise can significantly reduce simulation performance. Furthermore, both caches and the pipeline can be considered simultaneously during the PSTC, which enables an accurate consideration of their interactions that is not possible in a context-insensitive simulation. This applies to instruction caches in particular, as their impact on application timing is mainly influenced by the control flow. However, our experimental results demonstrate that it is also possible to accurately model the timing influence of data and unified caches if a dynamic PSTC is used. It remains to be seen how far these advantages can be maintained in multi- and many-core systems. For example, if coherent caches are used the timing of instructions executed on one core can be influenced by those executed on another core.

Another interesting advantage is offered by our simulation framework if a dynamic PSTC by hardware tracing is used. As the timing information is extracted from hardware, a software model of the hardware is not needed during the PSTC. Such a model is not only hard to construct, but the necessary details of the microarchitecture are not always publicly available.

20.5.2 Limitations

The main limitation of context-sensitive simulation is the need to re-execute the PSTC when the software code or the system configuration (to the extent considered in the PSTC) changes. This complicates the application of context-sensitive simulation in Design Space Exploration (DSE) and for self modifying code.

For systems with self-modifying code, for example, those using just-in-time compilation (e.g., Java in Android) or dynamic code relocation (e.g., shared libraries under Linux) context-sensitive simulation is currently not possible. However, for DSE, Plyaskin, Wild, and Herkersdorf [21] demonstrated that limited variations in the hardware architecture can still be considered in a context-sensitive simulation.

20.6 Conclusions

In this chapter we discussed context-sensitive timing simulation for embedded software. This concept improves simulation accuracy by considering the control flow that leads to the execution of an instruction sequence to more accurately approximate the timing of that particular execution. Furthermore, it permits novel approaches, such as highly accurate simulations of application performance over a wide range of inputs based on hardware measurements for only few inputs. Moreover, as could be observed in our experimental results, context-sensitive simulation can remove the need for slow online models of caches and branch prediction.

These advantages come at the drawback of requiring a PSTC of the application code before a simulation. This analysis induces an overhead which is only recouped if multiple and/or longer running simulations are performed based on the results of one analysis. Furthermore the complexity of dynamic and static PSTC complicates a practical application.

Even though these drawbacks appear significant, we expect an increase in the practical relevance of context-sensitive simulations in future hardware/software cosimulations. We base this assessment on three factors: First, future products such as autonomous vehicles will require highly complex and performance-demanding software which must fulfill high safety standards. Cosimulations using context-sensitive software timing simulation can provide a valuable tool to developers of these products. Second, we expect that future research will simplify the application of the approach, for example, by simplifying the PSTC by combining static and dynamic analysis techniques. Third, other future and existing simulation techniques using block-level timings can exploit contexts to improve simulation accuracy, even if they only apply simple mappings that only consider the preceding block.

References

1. AbsInt Angewandte Informatik GmbH (2016) aiT: worst-case execution time analyzers. http://www.absint.com/ait
2. ARM: CoreSight: V1.0 architecture specification
3. Bellard F (2005) QEMU, a fast and portable dynamic translator. In: Proceedings of the USENIX annual technical conference (ATEC)
4. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The GEM5 simulator. ACM SIGARCH Comput Archit News 39(2):1–7
5. Chakravarty S, Zhao Z, Gerstlauer A (2013) Automated, retargetable back-annotation for host compiled performance and power modeling. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS)
6. Chiang MC, Yeh TC, Tseng GF (2011) A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development. IEEE Trans Comput Aided Des Integr Circuits Syst 30(4):593–606
7. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of 4th ACM SIGACT-SIGPLAN symposium principles of programming languages
8. eCos. http://ecos.sourceware.org
9. Ferdinand C, Heckmann R, Langenbach M, Martin F, Schmidt M, Theiling H, Thesing S, Wilhelm R (2001) Reliable and precise WCET determination for a real-life processor. In: Embedded software. Springer, Berlin/Heidelberg, pp 469–485
10. Gustafsson J, Betts A, Ermedahl A, Lisper B (2010) The Mälardalen WCET benchmarks – past, present and future. In: Lisper B (ed) WCET2010. OCG, Brussels, pp 137–147
11. Kodak test images. http://www.cipr.rpi.edu/resource/stills/kodak.html
12. libjpeg-turbo. http://libjpeg-turbo.virtualgl.org/
13. Lu K, Müller-Gritschneder D, Schlichtmann U (2013) Fast cache simulation for host-compiled simulation of embedded software. In: Design, automation & test in Europe, pp 637–642. doi:10.7873/DATE.2013.139

14. Martin F, Alt M, Wilhelm R, Ferdinand C (1998) Analysis of loops. In: Compiler construction. Lecture notes in computer science, vol 1383. Springer, Berlin/Heidelberg, pp 80–94. doi:10.1007/BFb0026424
15. Nexus 5001 Forum (2012) Nexus 5001 Forum Standard. IEEE-ISTO 5001-2012
16. Ottlik S, Stattelmann S, Viehl A, Rosenstiel W, Bringmann O (2014) Context-sensitive timing simulation of binary embedded software. In: Proceedings of the 2014 international conference on compilers, architecture and synthesis for embedded systems, CASES'14
17. Ottlik S, Borrmann JM, Asbach S, Viehl A, Rosenstiel W, Bringmann O (2016) Trace-based context-sensitive timing simulation considering execution path variations. In: 21st Asia and South Pacific design automation conference (ASP-DAC)
18. Ottlik S, Gerum C, Viehl A, Rosenstiel W, Bringmann O (2017) Context-Sensitive Timing Automata for Fast Source Level Simulation, In: Proceedings of Design, Automation & Test in Europe (DATE), 2017
19. Plyaskin R, Herkersdorf A (2010) A method for accurate high-level performance evaluation of MPSoC architectures using fine-grained generated traces. In: Architecture of computing systems – ARCS 2010. Lecture notes in computer science, vol 5974. Springer, Berlin/Heidelberg, pp 199–210
20. Plyaskin R, Herkersdorf A (2011) Context-aware compiled simulation of out-of-order processor behavior based on atomic traces. In: 2011 IEEE/IFIP 19th international conference on VLSI and system-on-chip (VLSI-SoC)
21. Plyaskin R, Wild T, Herkersdorf A (2012) System-level software performance simulation considering out-of-order processor execution. In: 2012 international symposium on system on chip (SoC)
22. Rosa F, Ost L, Reis R, Sassatelli G (2013) Instruction-driven timing CPU model for efficient embedded software development using OVP. In: 2013 IEEE 20th international conference on electronics, circuits, and systems (ICECS)
23. Stattelmann S (2013) Source-level performance estimation of compiler-optimized embedded software considering complex program transformations. Verlag Dr. Hut, München
24. Stattelmann S, Bringmann O, Rosenstiel W (2011) Fast and accurate source-level simulation of software timing considering complex code optimizations. In: 2011 48th ACM/EDAC/IEEE design automation conference (DAC)
25. Stattelmann S, Ottlik S, Viehl A, Bringmann O, Rosenstiel W (2012) Combining instruction set simulation and WCET analysis for embedded software performance estimation. In: 2012 7th IEEE international symposium on industrial embedded system (SIES), pp 295–298
26. Thach D, Tamiya Y, Kuwamura S, Ike A (2012) Fast cycle estimation methodology for instruction-level emulator. In: 2012 design, automation & test in Europe conference & Exhibition (DATE)
27. Theiling H (2002) Control flow graphs for real-time systems analysis. Dissertation, Universität des Saarlandes

Part VI
Performance Estimation, Analysis, and Verification

Timing Models for Fast Embedded Software Performance Analysis

21

Oliver Bringmann, Christoph Gerum, and Sebastian Ottlik

Abstract

In this chapter, we give an overview on timing models which provide an abstract representation of the timing behavior for a given software. These models can be driven by a functional simulation based on the simulated control flow. As the timing model itself can reach a level of accuracy that is comparable to a classic timing simulation of the represented software, these approaches enable a fast yet accurate software performance analysis. In this chapter, we focus on the generation and structure of various models but also provide a brief introduction into their integration with a functional simulation. The presented approaches are targeting software executing on current and future system-on-chips with a wide range of embedded processors – including Graphics Processing Units (GPUs).

Acronyms

ASIP	Application-Specific Instruction-set Processor
CFG	Control-Flow Graph
CPU	Central Processing Unit
DSP	Digital Signal Processor
GPU	Graphics Processing Unit
MLoC	Million Lines of Code

O. Bringmann (✉)
Wilhelm-Schickard-Institut, University of Tübingen, Tübingen, Germany

Embedded Systems, University of Tübingen, Tübingen, Germany
e-mail: oliver.bringmann@uni-tuebingen.de

C. Gerum
Embedded Systems, University of Tübingen, Tübingen, Germany
e-mail: christoph.gerum@uni-tuebingen.de

S. Ottlik
Microelectronic System Design, FZI Research Center for Information Technology, Karlsruhe, Germany
e-mail: ottlik@fzi.de

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_22

RTL	Register Transfer Level
SoC	System-on-Chip
VIVU	Virtual Inlining and Virtual Unrolling
VLIW	Very Long Instruction Word
WCET	Worst-Case Execution Time

Contents

21.1 Introduction

The ever increasing demand for new and more advanced features in products including embedded systems is leading to an increased relevance and complexity of embedded software to be used to realize these features. In addition, the growing computational demand of embedded software can only be served by more and more complex hardware architectures. Furthermore, for many embedded systems, software performance or even strict adherence to timing requirements is a serious factor, in particular for safety-critical products. Therefore, it is essential to integrate efficient timing and performance analysis methods and tools into the development process.

Software timing simulations are one approach to software timing and performance analysis. In contrast to a direct evaluation of software on the target hardware, simulations offer many advantages; the most important are reproducibility and greatly enhanced observability. In contrast to static analysis, using simulation is more natural to a developer and does not suffer from the overly conservative approximations necessary to avoid state space explosion in static analysis.

Besides these decisive advantages, there are a number of factors that must be considered for simulation to achieve a high practical value: Firstly, a detailed simulation of the underlying hardware greatly impairs simulation performance, up to a degree where a simulation is essentially useless in many use cases. Therefore, an abstraction is necessary. Secondly, there is an inherent loss of simulation accuracy when raising the abstraction level. However, this accuracy loss needs to be kept within acceptable limits; otherwise, simulation results become meaningless. Thirdly, creating a complex simulation can take a considerable modeling effort. Consequently, when choosing a simulation approach, the time-to-model must be considered.

In this chapter, we discuss timing models that enable high-performance yet accurate timing simulation. Essentially, these models provide a target platform-specific model of the temporal behavior of the embedded software based on the internal control flow. As the control flow of software programs can be tracked very efficiently during fast functional simulation, these model can enable timing simulation with a very low overhead. Since many factors that influence software timing can be represented as a direct or an indirect function of the control flow, a very high simulation accuracy can be achieved. While these models can be generated automatically, additional knowledge of the target platform is required and needs to be modeled and generated.

A wide range of sources can be applied during model generation, ranging from observing code execution on prototyping hardware, where modeling effort is negligible, to complex analytical models used in abstract interpretation.

Use cases for these models can be mainly found in the domain of embedded software engineering and systems development. For example, they can be applied in early evaluations of nonfunctional properties or to select components in heterogeneous multiprocessor systems-on-a-chip. The need to regenerate the model when the target platform changes limits their use in microarchitecture design.

This chapter is organized as follows: In Sect. 21.2, we discuss software performance analysis in general but with a particular focus on simulation and a comparison of control-flow-driven timing models to other approaches. Afterward we give an overview of three different approaches to control-flow-driven timing models. These models differ in the necessary knowledge of the target platform and the modeling effort. In Sect. 21.3, we discuss an approach that allows target platform independent performance characterization during simulation. Target-specific timing estimates are then generated after the simulation using an analytical target model. In Sect. 21.4, we discuss an approach that employs a generic platform model, that only has to be parametrized for a given target platform. In Sect. 21.5, we discuss an approach that employs a specific platform model, that requires a detailed description of the target hardware. In Sect. 21.6, we focus on a case study on modeling GPU-based architectures as these kinds of architectures cannot be represented by related models. In Sect. 21.8, we compare the presented approaches to modeling the generation of software performance models. The main conclusions of this chapter are summarized in Sect. 21.9.

21.2 Background

21.2.1 Challenges in Performance Evaluation of Modern Embedded Systems

As the main challenges for performance modeling and evaluation, we see software complexity and code reuse, hardware complexity and heterogeneity, as well as speed of performance analysis, which are briefly discussed in the following.

21.2.1.1 Software Complexity and Code Reuse

Current embedded software is large and complex. For example, Boeings 787 is estimated to contain 6.5 Million Lines of Code (MLoC) for its avionics and on-board support systems, while current premium class automobiles even run more than 100 MLoC [8]. This high software complexity leads to an extensive reuse of software components across different products. Considering these complexities, performance modeling techniques need to be applicable to large software projects. In cases of code reuse in different target systems, it would be beneficial if the timing models could be reused if a software component is reused in a different product.

21.2.1.2 Hardware Complexity and Heterogeneity

Many embedded systems contain complex hardware architectures. The ARM processors from the Cortex-A series contain superscalar pipelines often with out-of-order execution. But not only complex general purpose Central Processing Units (CPUs) are used in current embedded systems, there are also a wide variety of specialized processors like Digital Signal Processors (DSPs), Application-Specific Instruction-set Processors (ASIPs), or Very Long Instruction Words (VLIWs) processors. Moreover, in recent years, different Graphics Processing Unit (GPU)-based architectures for embedded systems are offered [28] that combine standard embedded processor architectures with GPUs. These are suitable as accelerators for specific software tasks. Therefore, performance models should be able to represent the timing-relevant behavior of the components of these complex, heterogeneous systems.

21.2.1.3 Development Cycles and Modeling Effort

Embedded system designers face tight deadlines. This means that the development process needs to be parallelized and has to support timing analysis and timing error detection as early as possible. While hardware software codesign approaches allow to start software development before the developed hardware is available, performance models allow to expose performance relevant errors very early in the development of an embedded system [44].

To reach this goal, the effort for model generation should be as low as possible, and the generated models should be available very early in the development process.

21.2.1.4 Speed of Performance Analysis

When using timing models to evaluate the performance of embedded systems, the speed of performance analysis is always an important optimization goal. The application of high-level performance simulation increases the development productivity and acceptance as simulation can easily be integrated into the development process. There exist also some applications of performance models like *Software-in-the-Loop* simulation [47] where the performance simulation needs to be faster than the execution of the software on the target hardware.

21.2.2 Static Software Timing Analysis

Static software execution time analysis mostly uses a combination of abstract interpretation [11] and Programming (ILP) [48] to determine an estimate of the execution time of an embedded software without actually executing the software.

In static analysis, the modeling of timing behavior is only part of the analysis, while a reconstruction of program structure [26, 46], program values, and loop bounds is also an important aspect of the analysis. The usage of caches in embedded architectures further complicates the static timing analysis of embedded systems.

Static analysis has the advantage that, using properly designed analysis, certain properties of the timing estimation can be guaranteed. Especially a formally safe Worst-Case Execution Time (WCET) estimate is currently not possible using the other analysis techniques. On the other hand, static analysis is problematic as some architectures such as many-core processors and GPUs are currently not analyzable using a static analysis tool.

21.2.3 Simulation-Based Software Timing Analysis

In the following sections, we give an overview of methodologies that provide some notion of time while simulating the execution of software on a processor. In particular, we exclude trace-driven simulation [20] where only a prerecorded trace of instructions is replayed.

21.2.3.1 RTL Simulation

In principle, a simulation can be generated from the Register-Transfer Level (RTL) description of a system. While obtaining Register Transfer Level (RTL) code for a full system is not practical for most application developers, such models are sometimes available commercially. For example, ARM recently acquired Carbon Design System and plans to market models compiled from the RTL descriptions as ARM Cycle Models. However, even when accelerated using specialized compilers such as Verilator [19], these simulations have a very low performance and therefore are not a good choice for application performance analysis, unless exact results are absolutely required.

21.2.3.2 Fixed Throughput Simulation

Many commercial simulators, such as Imperas OVP [18] and ARM FastModels [1], as well as mainline QEMU, provide a simple timing simulation that is based on an user-specified instruction throughput. The main use case of these simulations is functional analysis, where the simplified timing simulation only serves to ensure a linear progression of time. As the throughput is assumed to be constant during a simulation but can vary significantly for nontrivial applications on real processors, these models are not appropriate for software performance analysis.

21.2.3.3 Microarchitectural Simulation

Microarchitectural simulation focuses on the interaction between hardware components of the system microarchitecture such as individual processor pipeline stages, functional units, or caches. Individual component models abstract implementation details and only aim at approximating timing characteristics. The simulation is usually cycle driven. For classical Systems-on-Chips (SoCs), well-known examples of this class of simulators are Simplescalar [2] and Gem5 [4]. GPGPUsim [3] applies this approach to GPUs.

While the low-abstraction level of these simulations suggest a near-exact simulation, this is usually not achieved. Reported simulation errors when modeling real processors [5, 16, 35] are, at best, equal to other approximate approaches, while performance is significantly lower [2, 10, 35]. The main benefit of these simulations is the ease of modifying low-level details (e.g., branch predictor policies). Therefore this approach is mainly useful for computer architecture research but not a good fit for software performance analysis.

21.2.3.4 Analytical Performance Estimation

Analytical performance models consist of a profiling phase and an estimation phase. In the profiling phase, performance metrics such as instruction count, cache miss rates, or the number of mispredicted branches are extracted during the execution of a program. During the estimation phase, the performance models then integrate the performance metrics to estimate program execution times using an analytic formula. Analytical models for performance estimations of GPU cores were presented in [22] and [31].

21.2.3.5 Phase-Based Performance Estimation

Another analytical approach is trying to reduce the simulation to phases of a program representative for the behavior of the whole program. Sampling-based analytical simulation models use a cycle accurate simulation but try to restrict the simulation to parts of a program. Very simple applications of this principle are just simulating the first n instructions of a program run and using the number of instruction per cycle obtained from this simulation to interpolate the timing behavior of the remainder of the program. A slight improvement can be reached when the representative phase from the middle of the execution trace [49].

An application of this simulation technique is SimPoint [37, 38]; it divides the program execution in phases of 100 million instructions. Phases are characterized

by the basic block vectors capturing the number of executions of each basic block during a phase. These basic block vectors are used to identify similar phases using a clustering algorithm. The performance of the programs is estimated from the results of a cycle accurate simulation of each phase closest to a cluster center. Other approaches combine a sampling-based model with a higher-level analytical performance model [43].

21.2.3.6 Interval-Based Simulation

Interval-based analytical performance models combine the profiling phase and the estimation phase in one run.

As shown in Fig. 21.1, these performance models divide the execution of a program in parts with a constant instruction throughput divided by stall events like branch mispredictions or cache misses. These models therefore allow a more accurate simulation of the interleaving in the processor pipeline compared to analytical models considering the whole program execution.

In simple interval-based models [12], the duration of an interval is approximated by the number of instructions in the interval divided by the dispatch width of the simulated processor pipeline and a miss penalty of the miss event at the end of the pipeline stage. Interval-based simulation is extended by Sniper [6] to improve the execution time estimation of an interval by incorporating information on the number of available functional units and allowing out of order execution of data cache miss events. GPUMech [17] is an interval-based GPU performance model. It uses interval-based simulation claims to simulate the timing behavior of GPU-based systems. Their methods are comparable to the ones used in Sniper for CPUs. This model has been applied to complex processor pipelines in [21].

21.2.3.7 Control Flow-Driven Simulation

Control flow-driven timing simulation is performed based on a priori estimations of the execution time for small portions of the program code, as shown in Fig. 21.2.

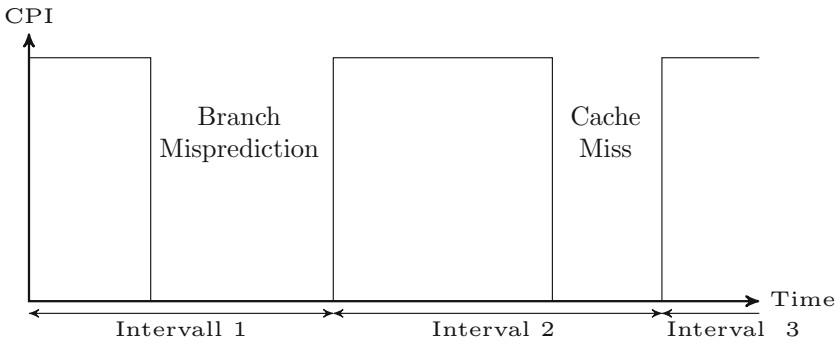


Fig. 21.1 Simulation by dividing time into intervals between miss events

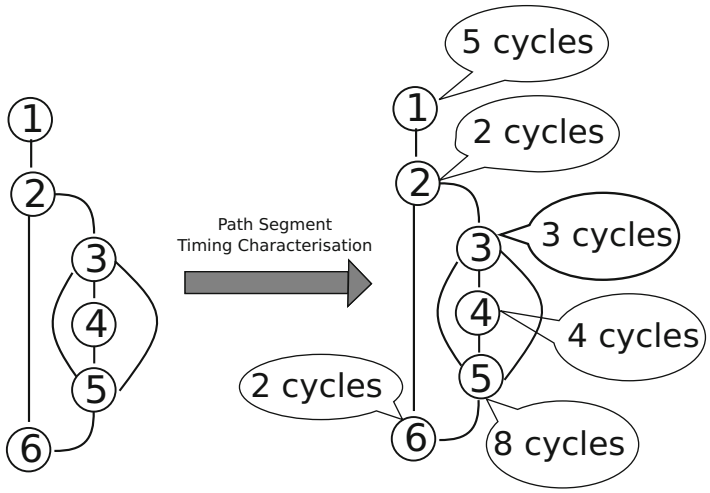


Fig. 21.2 Control flow-driven simulation uses static execution time estimates for small portions of programs

Most approaches use an a priori timing estimation to get the execution times for the basic block of the Control-Flow Graph (CFG).

For example, Tach, Tymiya, Kuwamura, and Ike [45] presented an approach where instruction block timings are first estimated assuming no cache misses or branch mispredictions. Timing is simulated by accumulating the block timings of all executed blocks and further penalties for cache misses and branch mispredictions.

Recent research [7, 13, 14, 29, 30, 32–34, 39, 41, 42] has demonstrated that simulation accuracy of this approach can be improved by differentiating different situations in which an instruction block is executed. In the most simple case [7, 13, 14], block timings are selected based on the preceding block. In contrast to a single block timing, this improves the consideration of instruction dependencies and instruction scheduling. Accuracy can be increased by considering more than one preceding block [33]. Besides an approach that implements this scheme, we also discuss a consideration of the preceding control flow using so-called Virtual Inlining and Virtual Unrolling (VIVU) context [29, 30, 42] in this chapter. These context enable a block timing granularity that allows an accurate simulation of complex applications on complex processor architectures without any online modeling of microarchitectural components, such as caches.

21.2.4 Summary

From the presented approaches especially control flow-driven simulation models combine many advantages. They clearly separate the functional simulation from the timing simulation. This allows a combination of these models with functional

simulation by source-level simulation (cf. ▸Chap. 17, "Parallel Simulation") or binary-level simulation (cf. ▸Chaps. 19, "Host-Compiled Simulation" and ▸20, "Precise Software Timing Simulation Considering Execution Contexts"). This separation also allows to choose different styles of timing models. In the following sections, we present three approaches to generate timing models for these kinds of systems at different abstraction levels.

21.3 Modeling Using Hardware-Independent Execution Cost Estimates

This approach to performance modeling of embedded systems tries to use the bare minimum of target specific information to produce a still meaningful result. While many approaches to generate performance models operate on the target-specific binary of a software, this approach aims to extract and quantify hardware-independent computational demand (HIC) from software source code and define a transition to gain hardware-specific execution costs (HSE). One main characteristic of our approach is that we extract computational demand for each software component from the source code. This has to be done only once. There is no need for target compilers or binary tools. We execute the application on the development platform to obtain data-dependent but hardware-independent execution characteristics of the application. If the developer changes components of the hardware platform or their configuration, only the transition to the HSE needs to be recalculated. The basic approach of the analysis is shown in Fig. 21.3.

The hardware independent computational demand is initially calculated on each basic block of the source-level control-flow graph of the original application. As

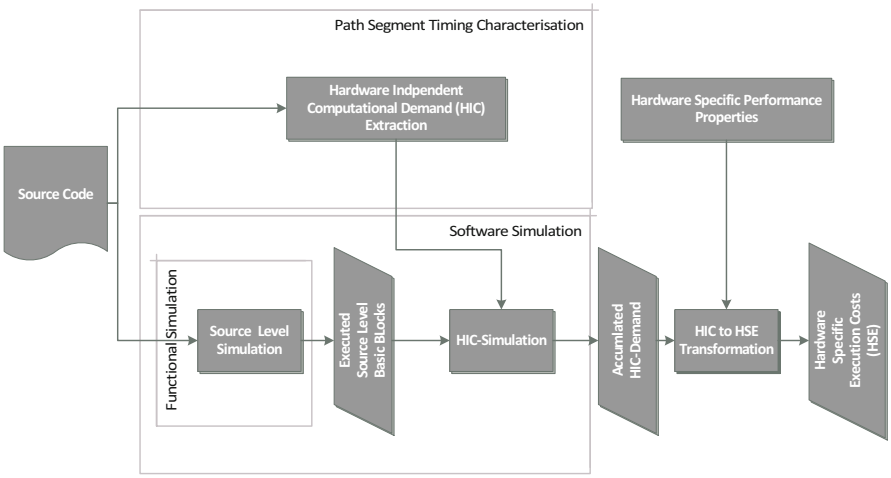
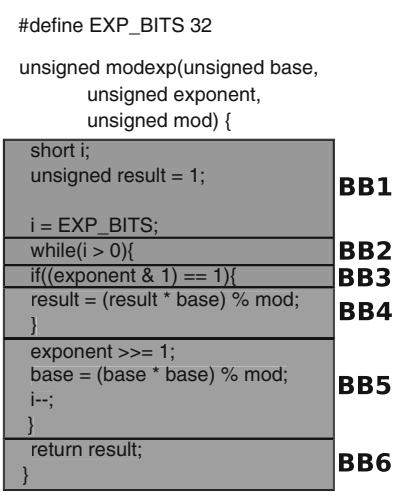


Fig. 21.3 Flow for the performance simulation with a hardware-independent computational demand simulation



computational demand		
BB1:		
= (int) 1	= (short) 1	
BB2:		
cond_branch 1	> (short) 1	
BB3:		
cond_branch 1	& (unsigned) 1	== (unsigned) 1
BB4:		
= (unsigned) 1	* (unsigned) 1	% (unsigned) 1
BB5:		
» 1	* (unsigned) 1	% (unsigned) 1
= (unsigned) 2	– (short) 1	
BB6:		
ind_branch 1		

Fig. 21.4 Calculating computational demand for a simple example

shown in Fig. 21.4, the approach extracts the computational demand by counting the operation type and the data types it operates on.

The HSE determination is based on abstract hardware specifications. The transition from HIC to HSE calculates the HSE for the considered pair of hardware and software. This estimation requires only an abstract hardware specification that can be extracted from the data sheets of the hardware platforms.

To calculate the hardware-specific execution costs, the target platform is specified with processor-specific and operation-specific attributes. The processor-specific attributes are:

1. The clock frequency at which the processor operates.
2. The branch predictor configuration, e.g., type and size of the target branch predictor
3. The used cache sizes and associativities and replacement strategies and cache miss penalties
4. A superscalar factor specifying the maximum number of instructions executed in parallel

In contrast to the general processor-specific attributes, the operation-specific attributes do not specify general behavior of the processor but give an execution cost attribute for the

1. The operation that is executed by this branch predictor.
2. The data type this operation operates on.
3. The number of cycles this operand needs for execution.
4. The number of parallel FUs that can execute this instruction type.

This means much less effort than implementing virtual models or using target compilers and binary tools for WCET analyzers. The transition only needs seconds which makes the approach much faster than the determination of HSE via virtual prototypes or ISS and can speed up the design process in complex software systems on heterogeneous hardware components before the initial mapping configuration is available. The HSE values can be used for an initial mapping determination approach. Experimental results show that the estimation is very fast and accurate enough to help designers making initial system configuration decisions.

21.4 Modeling Using Partial Architectural Knowledge

In this approach, we assume that the information available is comparable to the microarchitecture description given in architecture reference manuals of current microprocessors. This information is not sufficient to build a detailed cycle accurate simulator but can be used to derive analytic performance models of the microarchitecture. As generation of these kinds of models needs only partial knowledge of the modeled microarchitecture, we refer to them as partial microarchitectural models or microarchitecture-aware models.

The complete flow of the performance modeling and simulation using partial microarchitecture models is shown in Fig. 21.5. The method starts with the source code and the binary code of an application. We then extract the structure of the binary code CFG and do a structural matching of the source-level and binary-level control flow graphs and automatically annotated with function calls which reference the matched binary codes. Together with a generated path simulation code, this allows a simulation of binary-level paths through execution of the annotated source

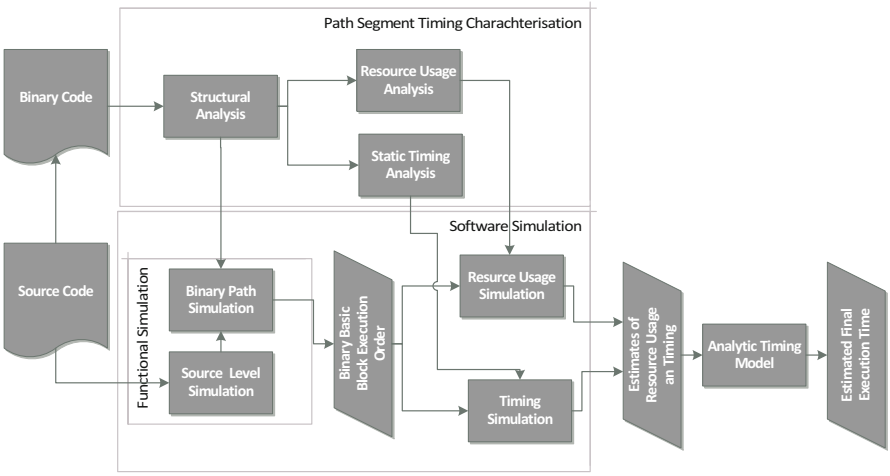


Fig. 21.5 Flow for the performance simulation with partial microarchitectural knowledge

code. The source to binary-level matching and path simulation code generation is not part of this chapter. Details of these steps are given in ▸ Chap. 17, "Parallel Simulation" and in [39–41].

The application-specific timing model is generated using an offline analysis of the binary-level control-flow graphs. It calculates for each basic block an estimated execution time depending on its predecessor. We additionally calculate a resource usage histogram for the instructions in each basic block. The results of the offline analysis are then added to the path simulation code and accumulate the per basic block matrix during execution of the instrumented source code. After the execution of the instrumented software, the accumulated performance metrics are used by a target-specific analytical performance model to produce a final estimate of the execution time.

Parts of the following subsections are based on our preceding publications [13, 14]. In these publications, we specifically focused on the application of this modeling style on GPUs. The following description is split in a part describing the modeling of GPU architectures (Sect. 21.6.1) and Sects. 21.4.1 and 21.4.2 describing the target agnostic parts of the model generation and simulation.

21.4.1 Static Timing Estimation Using Pipeline Execution Graphs

The static basic block timing analysis determines an optimistic execution time for each basic block in the binary-level control-flow graph. The effects of resource contention on the execution time can be incorporated by an analytical model after the timing simulation (Sect. 21.6.1). The analysis uses pipeline execution graphs [24] to model the timing behavior of each *instruction* on the pipeline. Our pipeline execution graph EG_B for a basic block is defined as

$$EG_B = (S_B, D_B, lat, use, res)$$

where the nodes in S_B represent each execution step for each instruction on the pipeline. $D_B \subset S_B \times S_B$ represents the dependence relation. It contains an edge for each dependence corresponding to the instructions in the basic block. In our model, an execution step might not directly correspond to a pipeline stage. The minimum latency between the start of an execution step and the start of a dependent execution step is given by the function $lat : D_B \rightarrow \mathbb{N}_0$. Latency is expressed in cycles.

To incorporate the dynamic resource usage into the timing model, we extend the pipeline execution graph model with an estimation of the resource usage. The resource usage function $use : V_B \rightarrow \mathbb{N}_0$ labels each step in the execution graph by the number of cycles the resources in this step are taken. The function $res : S_B \rightarrow \mathbb{N}_0$ maps each execution step to a unique identifier for the resource used in this step.

In Fig. 21.6, we show an example for a pipeline execution graph that is built for the analysis of an Bolero-3M-based microcontroller [36]. These microcontrollers are based on a PowerPC instruction-set architecture. Depending on the number of instructions in a PPC processor. The front end of the processor pipeline allows

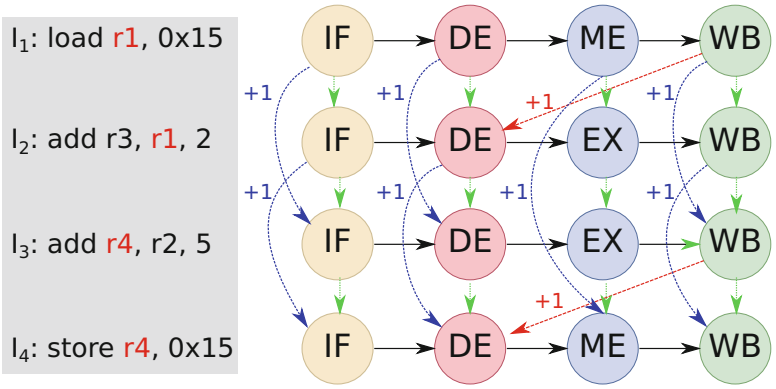


Fig. 21.6 Pipeline execution graph-based timing analysis

fetches of up to two 32-Bit instructions at once. This is indicated by the edges between the IF (instruction fetch) nodes of I_1 and I_3 and I_2 and I_4. So that every instruction is fetched one cycle after the instruction that is two positions ahead in the instruction stream. As the decode stage can handle two instructions as well, its representation is similar to the IF stage. In the third stage, the instructions are executed on multiple execution units. The instructions accessing memory are executed on a single memory unit (MEM), while there are two execution units for arithmetic and logic instructions available. This means there is a potential resource conflict between the MEM nodes of instruction I_1 and I_4. There is no potential resource conflict between I_1 and I_2 as the pipeline has two execution units for this instruction type available. The writeback to register file is handled for two instructions in parallel on this pipeline. The edges between the WB and DE nodes of the graph indicate a true data dependency. I1 loads data from memory that is used as an operand by the next instruction. In this case, the DE phase of instruction I_2 needs to wait for the cycle after the writeback of node instruction I_2. The same kind of dependency also exists between instruction I_3 and I_4.

Provided a pipeline execution graph, the timing analysis is done as a fixpoint iteration on the pipeline execution graph. The algorithm iterates over each state in the pipeline execution graph and calculates the earliest start times for each node by the maximum over the start times of the predecessors added with the corresponding latencies. The fixpoint iteration is finished when the start times for each state do not change anymore. To accelerate the convergence of the fixpoint iteration, we iterate over the states in topological sort order, but the result of the iteration is independent of the order in which the states are visited. The termination of the algorithm follows from the absence of cycles in the execution graph. For the static execution time analysis, we build the pipeline execution graph for the instructions of each basic block and calculate the fixpoint. It delivers a static timing analysis for the duration of the basic block by the maximum start time of each node in the execution graph. The analyzed execution time for a basic block v_i is called t_{v_i}. Building the execution

time of a kernel during simulation by summation over the basic block times t_i would overestimate the execution time of the kernel, because the execution of instructions from adjacent basic blocks can overlap. To incorporate this effect into the static timings, we also build pipeline execution graphs for the instructions from each pair of adjacent basic blocks. The analyzed execution time for each pair of basic blocks v_i, v_j is called $t_{(v_i,v_j)}$.

The results of this static analysis are then used for a simulative approximation of the timing behavior of a program.

21.4.2 Timing Annotation and Simulation

The timing information and resource information from static timing analysis is back annotated to the original source code, by inserting function calls to timing functions in the original source code and generating the corresponding timing function implementations. The algorithm for timing annotation is shown in Algorithm 1.

Algorithm 1 Algorithm for timing annotation

```
for v_S ∈ V_S do
  if ∃v_B ∈ V_B : map(v_B) = v_S then
    insert function call to timing simulation in v_S
    create function for timing simulation
    for paths p between a mapped block v'_B and v_B do
      Add code to function for:
      if last simulated block = v'_B then
        t_thread += Σ_{v_i ∈ p/v'_B} t_(v_{i-1},v_i) − t_{v_{i-1}}
        for resources r_i do
          u_thread(r_i) += Σ_{v_i ∈ p/v'_B} use(r_i, v_i)
        end for
      end if
    end for
  end if
end for
```

As the timing analysis has been done on the binary-level code and the simulation is based on annotations in the source code, the algorithm depends on a good mapping of binary basic blocks to source-level basic blocks. An example for a good method for mapping of source-level to binary-level control flow is given in ▸ Chap. 17, "Parallel Simulation". The algorithm first iterates over all mapped basic blocks in the source-level control-flow graph and inserts function calls that do the timing simulation according to the static analysis. The functions for timing simulation are generated in the inner loop of Algorithm 1. In this loop, the algorithm iterates over all binary-level paths between matched blocks and calculates the estimated execution time for this path by the sum over the pairwise execution times of the nodes in the path $t_{(v_{i-1},v_i)}$. As the summation would count the execution time for each start node twice, we subtract the execution time for each basic block $t_{v_{i-1}}$.

When doing simulations of GPU-based architectures, in addition to the execution times, we also annotate the threads resource usage for all resources in the pipeline. The results of an execution of the annotated source code on an OpenCL compatible device are an optimistic timing estimation of each thread and each accumulated resource usage of the thread for each resource in the pipeline. The final execution time of the kernel is estimated from these values using an analytical model. Details on an analytical model for GPUs are given in the following section.

21.5 Modeling Using Detailed Microarchitectural Knowledge

A complete, detailed description of the microarchitecture can be utilized to achieve an exact performance simulation. However, due to their extremely low performance, such simulations are, in practice, not a good choice for software performance analysis. Therefore, a simulation that achieves a high performance while maintaining an acceptable accuracy is preferable for this purpose, even if complete knowledge of the microarchitecture is available.

One concept for such a simulation is to shift most or all evaluations regarding the microarchitecture to an offline analysis that is executed before the simulation. The result of a single analysis can be reused by multiple simulations; the analysis cost is distributed among these simulations. This approach is therefore attractive for exploring software performance in different scenarios, such as over a wide range of input values. In this section, we give a brief overview of our simulation framework that realized this concept. More details can be found in ▸ Chap. 19, "Host-Compiled Simulation".

21.5.1 Framework Overview

An overview of our current framework is shown in Fig. 21.7. At first, the application binary code for the target processor is analyzed either statically or dynamically. In both cases, analysis results are stored in a software-specific timing model which is sometimes referred to as a timing database (TDB). A comparison of both analysis approaches is given in Sect. 21.5.2.

The timing model contains a description of the binary code control flow and timings for basic blocks of the program. Multiple timings are available for each block and differentiated by context. A context is an abstraction of the control flow leading to a block. The main advantages of using contexts is that the instructions that were executed before a block can be reflected in the context-dependent block timing. More details on this approach are given in Sect. 21.5.3.

A timing model can then be used in multiple simulations, where the timing simulation is driven by events indicating the execution of target basic blocks as defined by the control flow stored in the timing model. In a simulation of binary code execution (cf. ▸ Chap. 19, "Host-Compiled Simulation"), this can be achieved by instrumenting the first instruction of each basic block, whereas in a source-level

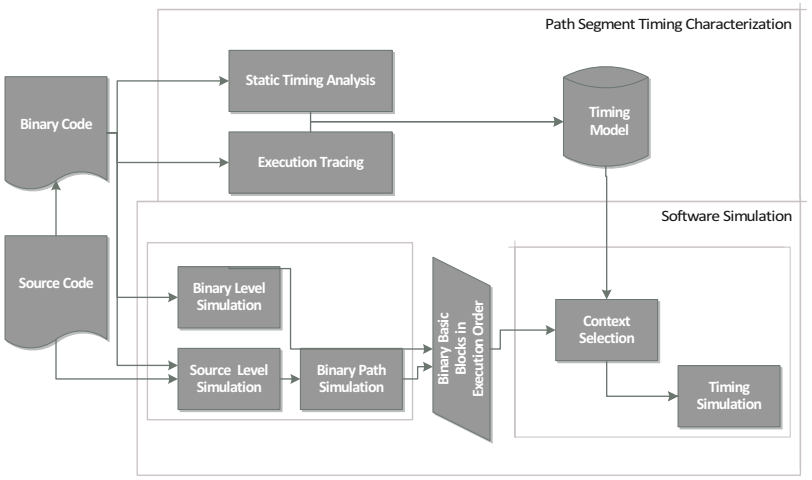


Fig. 21.7 Overview of the simulation framework

simulation (cf. ▸ Chap. 17, "Parallel Simulation", Section 21.4), a so-called path simulation is necessary, which simulates the target binary control flow based on the source-level control flow.

21.5.2 Static and Dynamic Analysis

The need for a precise microarchitectural model is the main drawback of detailed timing simulation. To limit the impact of this drawback, our framework is flexible in the kind of models that can be used by the offline analysis and supports both static and dynamic analyses. Thereby a model that was originally intended for a different purpose can be utilized by our simulation.

However, there are also various differences between static and dynamic analysis that lead to different advantages and disadvantages of one approach compared to the other. In principle, multiple timing models can be combined, which could also enable a hybrid analysis, but this topic is currently beyond the scope of our research.

In static analysis, the timing of the software is analyzed without executing the software. To avoid the halting problem, the actual states the software can get into have to be over-approximated. Firstly, this complicates the calculation of block timings. In practice, this currently restricts the application of static timing analysis to complex microarchitectures, in particular those including out-of-order execution. Secondly, it can lead to coverage deficits for programs containing asynchronous (e.g., interrupts) and indirect (e.g., function pointers) control flow changes. We developed a methodology to run multiple static analyses and combine their results to remove this issue.

In dynamic analysis, the timing of the software is analyzed by observing its execution. This avoids the issues of static analysis but makes it necessary to choose

program inputs that provide sufficient coverage. However, our experimental results demonstrate that selecting such inputs manually is feasible. Currently we observe software executions on target hardware, which removes the need for an additional model during the analysis.

21.5.3 Enhancing Accuracy by Considering Execution Contexts

On most modern embedded processors, the timing of an instruction sequence depends on the state of the microarchitecture before its execution, for example, due to instruction dependencies or cache contents. This state is heavily influenced by the already executed instructions. As aforementioned, this factor can be leveraged to improve simulation accuracy by obtaining multiple possible timings for each instruction block during the offline analysis and selecting an appropriate value for each execution of a block during the simulation.

More specifically, timings are differentiated by the preceding control flow. However, as the set of paths to block can in general be infinite and is likely excessively large for nontrivial programs, it is not possible to calculate a distinct timing for every path. Instead, the set of control flow paths is divided into a finite number of subsets, which are referred to as contexts, and block timings are differentiated by context.

In our framework, we apply the so-called VIVU contexts [26], which were originally developed for static analysis. Figure 21.8 shows a simple example for a VIVU contexts. This kind of context information captures the call stack and the loop iteration counts on the way from the start of the program to each basic block. The fact that this approach allows to distinguish the timing behavior of basic blocks depending on the call stack and the loop iteration count is reflected by its full name. Our experimental results demonstrate [30] that VIVU context coupled with a dynamic analysis by observing hardware executions enables a highly accurate timing simulation. This simulation is capable of simulating complex

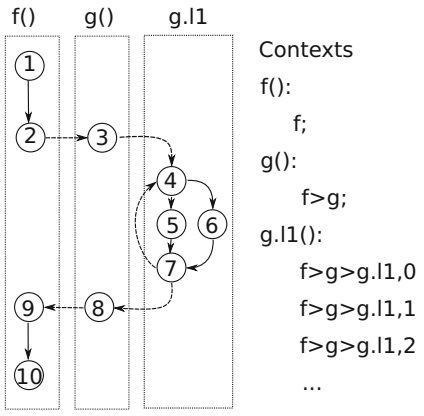


Fig. 21.8 An example for VIVU-context-based timing selection

software executing on complex processors at an error of typically less than 10% without any further modeling of microarchitectural elements including data caches.

21.6 Case Study: Modeling the Performance of a GPU-Based Microarchitecture

In this case study, we show the application of the timing model from Sect. 21.4 to a complex microarchitecture.

21.6.1 Applying the Simulation Approach to GPU Cores

To generate timing models for GPUs, the static analysis is run on the PTX assembly code and currently models the microarchitecture of a NVIDIA GTX 480 core. The analysis assumes that the pipeline is occupied by one warp exclusively.

As shown in Fig. 21.9, the static analysis models each instruction's execution on the given pipeline as a graph with five nodes. The first node (IF) corresponds to the front-end part of the pipeline up to the instruction buffer. The second part (IS) models the issue stage and the scoreboard. The latency of register accesses in the operand collector units is modeled by the node labeled OC. The fourth node (ALU) models the timing effects of the actual execution. This node is labeled according to the used execution unit (ALU, SFU, MEM). The last node models the writeback stage of the pipeline. This node is labeled WB.

Figure 21.10 shows an example of a pipeline execution graph for three add instructions on the pipeline of a GeForce GTX480 which is quite similar to the GPU core used in NVIDIAs embedded SoC Tegra K1 [25]. The latencies inherent to the

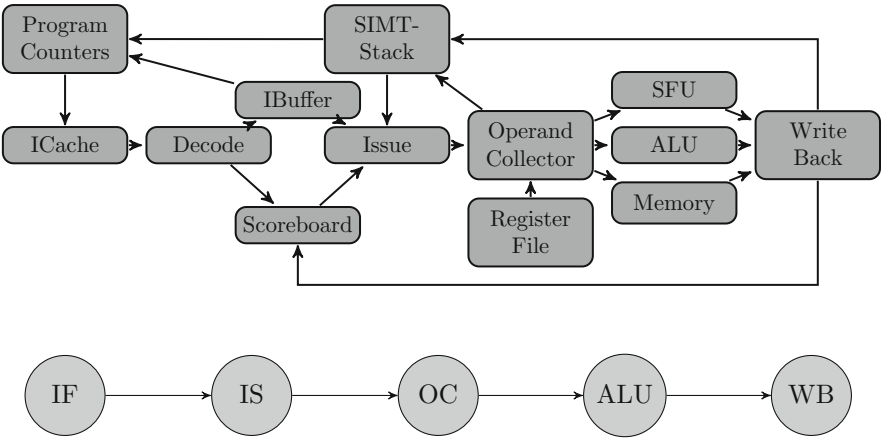


Fig. 21.9 GPU microarchitecture and the pipeline model

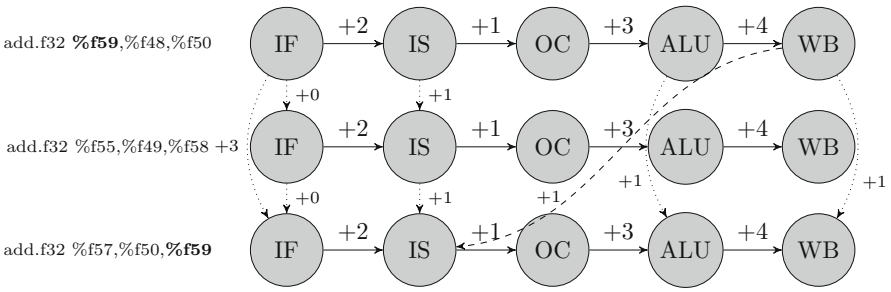


Fig. 21.10 Example of our pipeline analysis

pipelined execution of each instruction on the pipeline are shown by black edges. Instructions always take two cycles from instruction fetch to issue. The latency from issue to the operand collectors is one cycle for all instructions. The latency of the operand collectors depends on the number of registers read by the operation. The best case is always the number of registers read by the operation. The latency from execute to writeback can vary greatly depending on the instruction type. Load/store instructions show one cycle under the assumption of a cache hit and full coalescing of memory accesses, while double precision floating point division has a latency of 330 cycles. Resource dependencies between instructions are modeled by the blue edges in Fig. 21.10. As the front end fetches two adjoining instructions at a time in program order, each instruction is connected by an edge with latency zero in program order. To integrate the additional latency that every instruction is only fetched when the two entry instruction buffer is empty, we add additional edges between every second instruction. These edges have a latency of three cycles as this is the best-case instruction fetch latency. The issue stage issues one instruction a cycle in program order; this is modeled by an edge with latency one between each successive instruction. Resource dependencies in the execute stage are modeled in the same way. If there are multiple copies of a resource, the modeled pipeline has two ALUs, and the resource dependency edges skip instructions to account for the multiplicity of the resource. The same applies to the writeback step, as the pipeline can writeback two results at a time, there is only a dependency between every second node in WB. Data dependencies are always modeled by an edge with latency one from the writeback of the preceding instruction to the issue state of the depending instruction.

21.6.1.1 Analytical Timing Approximation for GPUs

Due to the inherent parallelism of GPU architectures, the cycle times and resource usages obtained from executing the annotated source code are specific to individual threads of the executed kernel, the levels of parallelism handled within the pipeline of a GPU core, simultaneous multi-threading, and warp-level parallelism.

Warps consist of the instructions of multiple threads from the same local work group. While the warp size may vary depending on the size of local work groups,

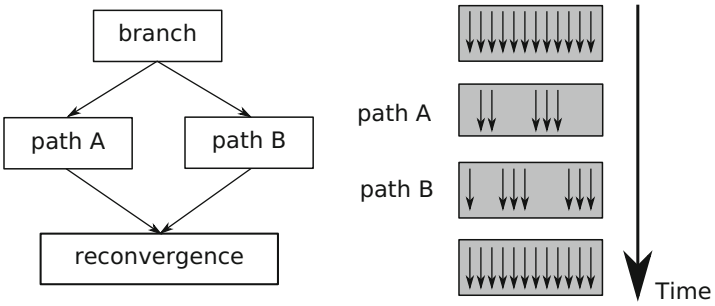


Fig. 21.11 An example of branch divergence

the preferred and maximum warp size on NVIDIA GPUs has been 32 threads for several generations. The threads of a warp always execute the same instruction in lockstep but are allowed to branch independently. As the instructions are allowed to branch independently, a so-called branch divergence can occur. Branch divergence is handled in hardware by executing each path sequentially and masking those operations which did not take the currently selected path. Figure 21.11 shows an example of branch divergence.

Apart from branch divergence, warps execute the instructions of several threads in lockstep. Each streaming multiprocessor handles multiple warps concurrently using fine-grained multi-threading.

The timing model uses the per thread execution times and resource usages as determined by the native execution of the annotated source code to calculate an estimate of the execution time of the whole kernel. The analytical model follows the hierarchy of parallelism of the multi-threaded execution. The algorithm starts with the execution time of the threads as determined by the source-level simulation. The execution time of a warp is then calculated from the execution time of the threads that form this warp. The execution times of all warps in a work group are combined to form the execution time of a work group. The execution time of the whole kernel is then estimated from the execution times of all work groups in the kernel. Timings up to this point do not consider the resource contention due to parallel execution of warps in the pipeline. This resource contention is taken into account by a final correction step.

The first step of the timing estimation calculates the execution time of each warp. This is done by first determining which threads form a common warp and then calculating the execution time of a warp t_{W_i} by taking the maximum execution time of all threads in the warp.

$$t_{W_i} = \max_{t_i \in W_i}(t_{t_i})$$

This calculation is motivated by the fact that all threads in a warp are executing the same instruction in lockstep. Due to branch divergence, it is still possible that the

simulated execution time of the threads in warp shows different execution times. The effects of branch divergence are approximated by taking the maximum execution time of all threads in the warp. This does not fully simulate all timing effects of branch divergence but accurately handles the most important case, of branch divergence at an *if* statement without an *else* or branch divergence at a loop exit condition. Real branch divergence in an if-else statement is not fully handled by our current model, but taking the maximum execution time still approximates the timing effects of this case. The execution times of a local work group WG_j is modeled by the end time of the last warp t_{last_j} in the work group. All warps in a local work group are started at the same time. So we can calculate the end time of the last warp by taking the maximum execution time of all warps in the local work group.

$$t_{last_j} = \max_{t_{w_i} \in WG_j} (t_{w_i})$$

Due to the limited bandwidth of the issue stage, the threads of a local work group, the i-th warp of a local work group issues at least $\lfloor i/2 \rfloor$ cycles after the first thread of the local work group. To incorporate this in our model, we add this to the finish time of the last warp. This leads us to the final equation for the execution time of the last warp

$$t_{last_j} = \max_{t_{w_i} \in WG_j} (t_{w_i} + \lfloor i/2 \rfloor)$$

The finish time of the last warp in each local work group is used to calculate the execution times of a kernel. The local work groups of a kernel can be executed in parallel, but due to constraints of a GPU's hardware, not all local work groups might be able to run in parallel. Given the number of parallel work groups n_{par}, we estimate the finish time of each work group. The first n_{par} work groups are started in parallel. The next work group is started when a work group finishes. This is expressed by the following equation:

$$t_{WG_j} = \begin{cases} t_{last_j} & : j < n_{par} \\ \min_{k \in \{j-n_{par},\ldots,j-1\}}(t_{WG_k}) + t_{last_j} & : j \geq n_{par} \end{cases}$$

The upper part of the equation calculates the finish times for the first n_{par} work groups, by using the finish time of the last warps. All further work groups are simulated by taking the minimum over the n_{par} predecessors and adding the finish time of the last warp in this work group. The optimistic execution time of the whole kernel is then the maximum finish time over all work groups:

$$t_{kernel\_opt} = \max_{WG_i}(t_{WG_i})$$

The kernel execution time so far does not consider any delays due to resource conflicts between multiple warps on the same pipeline. These resource conflicts

are modeled by the last step of our analytical model. The analytical model first calculates the resource usage $use_{W_j}(r_i)$ of a warp W_j as the maximum resource usage over all threads in the warp:

$$use_{W_j}(r_i) = \max_{t_k \in W_j} (use_{t_k}(r_i))$$

The resource usage is calculated for each resource r_i. We then calculate the resource usage of the whole kernel by summation over the resource usage of all warps in the kernel:

$$use_{kernel}(r_i) = \sum_{W_i \in Warps} use_{W_i}(r_i)$$

The optimistic execution time is then combined with the kernel's maximum resource usage to form the final execution time estimate. The most successful combination of resource usage and optimistic execution time we have found is the maximum of both values.

$$t_{kernel} = \max(t_{kernel\_opt}, \max_{r_i \in R} use_{kernel}(ri))$$

This model is surprising as it only takes resource contention into account when it is certain that there must be resource conflicts in the pipeline. But as GPUs are optimized for a high throughput, this model seems a reasonable choice. The accuracy of these models can be improved by doing a probabilistic approximation of the resource conflicts [14].

21.6.2 Results

We evaluated the performance model using several synthetic and real world benchmarks. All simulations were run on an Intel Core i7-4770 K CPU at 3.5 GHz with a NVIDIA GTX780 GPU with 12 streaming multiprocessors at 952 MHz. For the execution of the instrumented source code, we used either the CPU using Intel's implementation of OpenCL for CPUs or on the GPU using NVIDIA's implementation of OpenCL for GPUs. For comparison, we used the cycle accurate GPU simulator *gpgpu-sim*. Our configuration is based on the configuration for a NVIDIA GTX480 GPU as delivered by gpgpu-sim, but we reduced the model to more closely resemble an NVIDIA embedded gpu core. We also activated the so-called perfect memory mode of gpgpu-sim. This mode handles all memory accesses as cache hits. We choose to use a simulator for our evaluation as only limited information on the internal architecture of actual GPUs is available, and we needed to verify the results of the pipeline model ignoring influences from the memory subsystem.

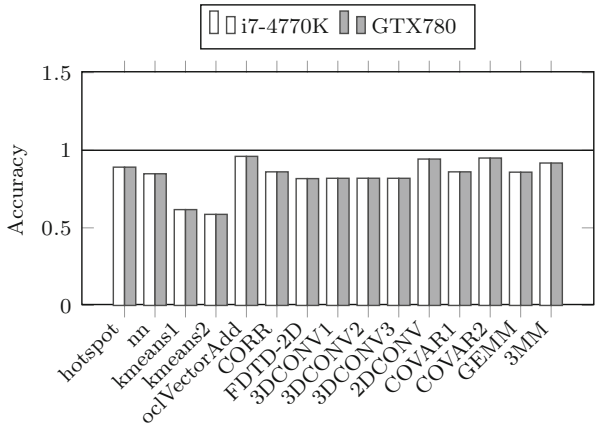


Fig. 21.12 Accuracy of the GPU performance simulation

The proposed simulation framework has been implemented as a shared library implementing the run-time API for translation and running of OpenCL kernels. For performance simulations, the library can be preloaded using the standard Unix `LD_PRELOAD` mechanism, so no changes to the host binaries used for simulation are needed. The kernels were translated to PTX code using *clang* as compiler [23]. All benchmarks were run with most compiler optimizations enabled (`-O3`), and we used the compiler switch to enable debug information in the compiler-generated assembly files (`-g`). We evaluated the performance model with benchmarks from the Rodina [9] and polybench-gpu [15] Benchmarks.

Figure 21.12 shows the execution times as estimated by our method divided by the execution times provided by `gpggpu-sim`. All our execution times underestimate the execution time as is expected by the best-case assumptions made by the performance modeling. For all but two kernels, the proposed simulation technique provides an accuracy of 80% or higher.

The accuracy results do not change between execution of the instrumented source code on a CPU or GPU as the performance simulation can be run on any OpenCL compatible device. The speedups in terms of the simulation time are shown in Fig. 21.13. The simulation time of our tool includes the time for data transfers from and to the OpenCL device, the execution of the kernel on the device, and the execution time of the analytical resource conflict model. As our goal is to support the simulation of long running application scenarios, the speedups do not include the time used for the static analysis of GPU kernels and binary to source matching. When instrumented source code is run on a CPU, speedups range between 140 for *oclVectorAdd* and 24061 for *kmeans2*. If the instrumented source code is run on a GPU, the speedups range between 477 for *oclVectorAdd* and 67750 for *COVAR*. The variation of execution speeds is partly explained by different basic block sizes in the applications. The other important factor considering speedups is the proportion of execution time of a thread to the number of threads. In our

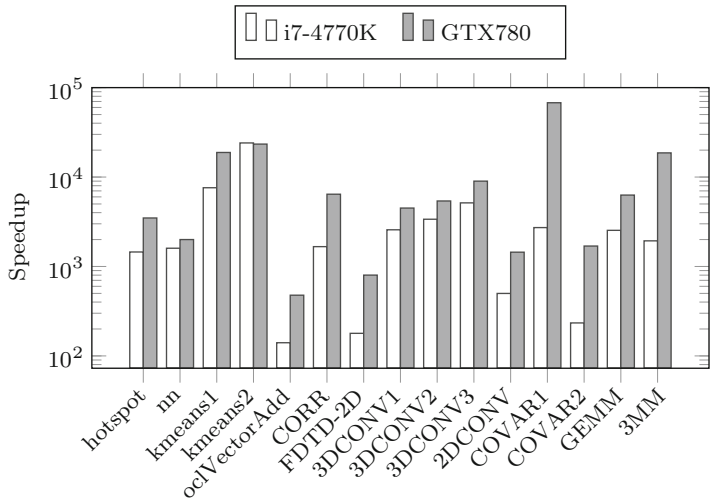


Fig. 21.13 Speedup of the simulation compared to the ISS *gpgpusim*

model, threads are simulated with the parallelism of the OpenCL device used for the simulations but our analytical performance model is run on the host without the use of parallelism. All benchmarks except *kmeans1* show an improvement of the simulation speed when the instrumented source code is executed on the GPU. The simulation performance of *kmeans1* degrades slightly as this benchmark does not fully utilize the available parallelism on the GPU.

21.7 Approaches to Include a Cache and Memory Simulation

The models so far assume that the latency of instructions can be statically determined. There are three basic approaches to integrate the simulation.

The simplest approach is an offline cache simulation. Prerequisite for an offline cache simulation is an approximation prior to the performance analysis. In an offline cache simulation, the cache miss rates of the application can be estimated using an external cache simulator. Either by executing the target binary code of the application on an instruction-set simulator or by executing the application on a host-compiled cache simulator like *cachegrind* [27]. Cache miss rates might even be provided as estimates by the developer using the performance model. The effect of the memory performance can then be incorporated into the model generation by increasing the latencies for each memory accessing instruction using an adoption of the standard formula to approximate the average memory access time:

$$\text{Average Memory Access Time} = \text{Cache Hit Time} + \text{Cache Miss Rate} \cdot \text{Miss Penalty}$$

This is the approach that is most compatible with the hardware-independent computational demand simulation, as the average memory access time can be used to dynamically determine the hardware-independent computational demand.

In contrast to an offline analysis that is run ahead of the simulation time, an online analysis that is run concurrently to the timing simulation might be carried out.

In the case of context-sensitive models based on hardware tracing, the memory subsystem often does not need to be modeled, as the timing impact of the memory subsystem is already part of the timed basic block traces, and a context-sensitive performance model reflects the timing impact well enough for many applications.

When higher accuracies or safe bounds are required. The timing relevant behavior of the memory subsystem can be analyzed by using a fully static analysis. In this case, the cache analysis is run in before the pipeline analysis and the results of the cache analysis are integrated with the pipeline analysis by changing the latencies of each individual instruction. If context-sensitive analyses are used, the pipeline analyses use individual latencies for each pair of instruction and context.

21.8 Discussion

The modeling techniques presented in this chapter represent models at different levels of abstraction and different trade-offs considering the challenges from Sect. 21.2. In this section, we briefly compare different modeling techniques addressing these challenges.

21.8.1 Comparison of Modeling Techniques

All of the considered modeling techniques have some specific drawbacks. For early platform component selection and task mapping onto different cores, it is probably the best solution to use hardware-independent execution cost estimates taken from Sect. 21.3 as these models require the least development effort and allow a good performance approximation for a wide variety of hardware components. However, the main drawback of these models are their quite low simulation accuracy.

The highest accuracy can be achieved by using context-sensitive performance models with detailed knowledge about the underlying microarchitecture (Sect. 21.5). These models often come up with an average simulation error below 1% and a maximum simulation error below 10%. The main drawback of this technique is the modeling effort and the need of a detailed microarchitectural model in terms of a cycle accurate simulator, a static WCET analysis tool, or a hardware implementation using a complex tracing unit. At least one of these approaches is generally available for standard embedded CPUs, but these are not generally available for application-specific processors or GPUs. If none of these approaches are available, the generation of a performance model would become necessary which requires an effort of several person months.

The technique using partial knowledge about the microarchitecture (Sect. 21.4 tries to balance the modeling effort and the model accuracy. Timing models can be developed in less than a month, and these models can represent most of the microarchitectural timing behavior of modern pipelined architectures. So these models are available very early on in the design process. This modeling style is also the only one that has been successfully applied to GPU-based microarchitectures, making it currently the correct choice for modeling of application-specific processors especially GPUs and GPU-based architectures.

21.9 Conclusions

In this chapter, we have discussed three different approaches to provide models for software performance analysis and compared their drawbacks. None of these techniques are able to fulfill all characteristics desired from a software performance model. Future research will need to allow an easier integration and seamless switching between different modeling techniques. Furthermore, an easy optimization of a performance model toward a timing characteristics of a desired target application has to be considered.

References

1. Arm fast models. http://www.arm.com/products/tools/models/fast-models/
2. Austin T, Larson E, Ernst D (2002) Simplescalar: an infrastructure for computer system modeling. Computer 35(2):59–67
3. Bakhoda A, Yuan GL, Fung WWL, Wong H, Aamodt TM (2009) Analyzing CUDA workloads using a detailed GPU simulator. IEEE, pp 163–174. doi:10.1109/ISPASS.2009.4919648
4. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The Gem5 simulator. SIGARCH Comput Archit News 39(2):1–7. doi:10.1145/2024716.2024718
5. Butko A, Garibotti R, Ost L, Sassatelli G (2012) Accuracy evaluation of Gem5 simulator system. In: 2012 7th international workshop on reconfigurable communication-centric systems-on-Chip (ReCoSoC), pp 1–7. doi:10.1109/ReCoSoC.2012.6322869
6. Carlson TE, Heirman W, Eyerman S, Hur I, Eeckhout L (2014) An evaluation of high-level mechanistic core models. ACM Trans Archit Code Optim (TACO). doi:10.1145/2629677
7. Chakravarty S, Zhao Z, Gerstlauer A (2013) Automated, retargetable back-annotation for host compiled performance and power modeling. In: Proceedings of the ninth IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis (CODES+ISSS), Newport Beach
8. Charette RN (2009) This car runs on code. IEEE Spectr 46(3):3
9. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE international symposium on workload characterization (IISWC), vol 2009. IEEE, pp 44–54. doi:10.1109/IISWC.2009.5306797
10. Chiang MC, Yeh TC, Tseng GF (2011) A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development. IEEE Trans Comput Aided Des Integr Circuits Syst
11. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium principles of programming languages, New York

12. Eyerman S, Eeckhout L, Karkhanis T, Smith JE (2009) A mechanistic performance model for superscalar out-of-order processors. ACM Trans Comput Syst 27(2):3:1–3:37. doi:10.1145/1534909.1534910
13. Gerum C, Bringmann O, Rosenstiel W (2015) Source level performance simulation of GPU cores. In: Design automation and test Europe, Grenoble
14. Gerum C, Rosenstiel W, Bringmann O (2015) Improving accuracy of source level timing simulation for GPUs using a probabilistic resource model. In: International conference on embedded computer systems: architectures modeling and simulation (SAMOS), Samos
15. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to GPU codes. In: 2012 innovative parallel computing (InPar). IEEE, pp 1–10. doi:10.1109/InPar.2012.6339595
16. Gutierrez A, Pusdesris J, Dreslinski RG, Mudge T, Sudanthi C, Emmons CD, Hayenga M, Paver N (2014) Sources of error in full-system simulation. In: 2014 IEEE international symposium on performance analysis of systems and software (ISPASS). IEEE, Piscataway, pp 13–22
17. Huang JC, Lee JH, Kim H, Lee HHS (2014) GPUMech: GPU performance modeling technique based on interval analysis. In: 47th annual IEEE/ACM international symposium on microarchitecture, pp 268–279. doi:10.1109/MICRO.2014.59
18. Imperas open virtual platforms. http://www.ovpworld.org/
19. Introduction to verilator. http://www.veripool.org/wiki/verilator
20. Isshiki T, Li D, Kunieda H, Isomura T, Satou K (2009) Trace-driven workload simulation method for multiprocessor system-on-chips. In: Proceedings of the 46th annual design automation conference, San Francisco
21. Karkhanis TS, Smith JE (2004) A first-order superscalar processor model. SIGARCH Comput Archit News 32(2):338–349. doi:10.1145/1028176.1006729
22. Lai J, Seznec A (2012) Break down GPU execution time with an analytical method. ACM Press, New York, pp 33–39. doi:10.1145/2162131.2162136
23. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 international symposium on code generation and optimization (CGO'04), Palo Alto
24. Li X, Roychoudhury A, Mitra T (2006) Modeling out-of-order processors for WCET analysis. Real-Time Syst 34(3):195–227. doi:10.1007/s11241-006-9205-5
25. Li A, Serban R, Negrut D (2014) An overview of NVIDIA Tegra K1 architecture. http://sbel.wisc.edu/documents/TR-2014-17.pdf
26. Martin F, Alt M, Wilhelm R, Ferdinand C (1998) Analysis of loops. In: compiler construction. Lecture notes in computer science, vol 1383. Springer, Berlin/Heidelberg, pp 80–94. doi:10.1007/BFb0026424
27. Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of ACM SIGPLAN conference on programming language design and implementation (PLDI), Seattle
28. Nvidia: NVIDIA Tegra K1 A New Era in Mobile Computing, pp 1–26
29. Ottlik S, Stattelmann S, Viehl A, Rosenstiel W, Bringmann O (2014) Context-sensitive timing simulation of binary embedded software. In: Proceedings of the 2014 international conference on compilers, architecture and synthesis for embedded systems (CASES)
30. Ottlik S, Borrmann JM, Asbach S, Viehl A, Rosenstiel W, Bringmann O (2016) Trace-based context-sensitive timing simulation considering execution path variations. In: 21st Asia and South Pacific design automation conference (ASP-DAC), Hong Kong
31. Parakh AK, Balakrishnan M, Paul K (2012) Performance estimation of GPUs with cache. IEEE, pp 2384–2393. doi:10.1109/IPDPSW.2012.328
32. Plyaskin R, Herkersdorf A (2010) A method for accurate high-level performance evaluation of MPSoC architectures using fine-grained generated traces. In: Architecture of computing systems – ARCS 2010. Lecture notes in computer science, vol 5974. Springer, Berlin/Heidelberg, pp 199–210

33. Plyaskin R, Herkersdorf A (2011) Context-aware compiled simulation of out-of-order processor behavior based on atomic traces. In: 2011 IEEE/IFIP 19th international conference on VLSI and system-on-Chip (VLSI-SoC), Hong Kong
34. Plyaskin R, Wild T, Herkersdorf A (2012) System-level software performance simulation considering out-of-order processor execution. In: 2012 international symposium on system on chip (SoC), Tampere
35. Rosa F, Ost L, Reis R, Sassatelli G (2013) Instruction-driven timing CPU model for efficient embedded software development using OVP. In: 2013 IEEE 20th international conference on electronics, circuits, and systems (ICECS), Abu Dhabi
36. Semiconductor F, Microelectronics S (2012) Bolero_3m microcontroller reference manual
37. Sherwood T, Perelman E, Calder B (2001) Basic block distribution analysis to find periodic behavior and simulation points in applications. In: Proceedings of the 2001 international conference on parallel architectures and compilation techniques. IEEE, Washington
38. Sherwood T, Perelman E, Hamerly G, Calder B (2002) Automatically characterizing large scale program behavior. ACM SIGARCH Comput Archit News 30(5):45–57, ACM
39. Stattelmann S (2013) Source-level performance estimation of compiler-optimized embedded software considering complex program transformations. Verlag Dr. Hut
40. Stattelmann S, Bringmann O, Rosenstiel W (2011) Dominator homomorphism based code matching for source-level simulation of embedded software. In: Proceedings of the seventh IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES+ISSS'11. ACM, New York, pp 305–314. doi:10.1145/2039370.2039417
41. Stattelmann S, Bringmann O, Rosenstiel W (2011) Fast and accurate source-level simulation of software timing considering complex code optimizations. In: Proceedings of the 48th design automation conference (DAC), San Diego
42. Stattelmann S, Ottlik S, Viehl A, Bringmann O, Rosenstiel W (2012) Combining instruction set simulation and WCET analysis for embedded software performance estimation. In: 2012 7th IEEE international symposium on industrial embedded system (SIES), Karlsruhe, pp 295–298
43. Van den Steen S, De Pestel S, Mechri M, Eyerman S, Carlson T, Black-Schaffer D, Hagersten E, Eeckhout L (2015) Micro-architecture independent analytical processor performance and power modeling. In: 2015 IEEE international symposium on performance analysis of systems and software (ISPASS), pp 32–41. doi:10.1109/ISPASS.2015.7095782
44. Teich J (2012) Hardware/software codesign: the past, the present, and predicting the future. Proc IEEE 100(Special Centennial Issue):1411–1430
45. Thach D, Tamiya Y, Kuwamura S, Ike A (2012) Fast cycle estimation methodology for instruction-level emulator. In: 2012 design, automation & test in Europe conference & exhibition (DATE), Dresden
46. Theiling H (2002) Control flow graphs for real-time systems analysis. Dissertation, Universität des Saarlandes
47. Werner S, Masing L, Lesniak F, Becker J (2015) Software-in-the-loop simulation of embedded control applications based on virtual platforms. In: 2015 25th international conference on field programmable logic and applications (FPL). IEEE, Piscataway, pp 1–8
48. Wilhelm R (2004) Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In: Proceedings of the 5th international conference on verification, model checking, and abstract interpretation, VMCAI 2004, Venice, pp 309–322. doi:10.1007/978-3-540-24622-0_25
49. Yi JJ, Kodakara SV, Sendag R, Lilja DJ, Hawkins DM (2005) Characterizing and comparing prevailing simulation techniques. In: 11th international symposium on high-performance computer architecture (HPCA-11), San Francisco

Semiformal Assertion-Based Verification of Hardware/Software Systems in a Model-Driven Design Framework

22

Graziano Pravadelli, Davide Quaglia, Sara Vinco, and Franco Fummi

Abstract

Since the mid-1990s, Model-Driven Design (MDD) methodologies (Selic, IEEE Softw 20(5):19–25, 2003) have aimed at raising the level of abstraction through an extensive use of generic models in all the phases of the development of embedded systems. MDD describes the system under development in terms of abstract characterization, attempting to be generic not only in the choice of implementation platforms but even in the choice of execution and interaction semantics. Thus, MDD has emerged as the most suitable solution to develop complex systems and has been supported by academic (Ferrari et al., From conception to implementation: a model based design approach. In: Proceedings of IFAC symposium on advances in automotive control, 2004) and industrial tools (3S Software CoDeSys, 2012. http://www.3s-software.com; Atego ARTiSAN, 2012. http://www.atego.com/products/artisan-studio; Gentleware Poseidon for UML embedded edition, 2012. http://www.gentleware.com/uml-software-embedded-edition.html; IAR Systems IAR visualSTATE, 2012. http://www.iar.com/Products/IAR-visualSTATE/; rhapsodyIBM Rational Rhapsody, 2012. http://www.ibm.com/software/awdtools/rhapsody; entarchSparx Systems Enterprise architet, 2012. http://www.sparxsystems.com.au; Aerospace Valley TOPCASED project, 2012. http://www.topcased.org). The gain offered by the adoption of an MDD approach is the capability of generating the source code implementing the target design in a systematic way, i.e., it avoids the need of manual writing. However, even if MDD simplifies the design implementation, it does not prevent the designers from wrongly defining the design behavior. Therefore, MDD gives full benefits if it also integrates *functional verification*.

G. Pravadelli (✉) • D. Quaglia • F. Fummi
Università di Verona, Verona, Italy
e-mail: graziano.pravadelli@univr.it; davide.quaglia@univr.it; franco.fummi@univr.it

S. Vinco
Politecnico di Torino, Turin, Italy
e-mail: sara.vinco@polito.it

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_23

In this context, Assertion-Based Verification (ABV) has emerged as one of the most powerful solutions for capturing a designer's intent and checking their compliance with the design implementation. In ABV, specifications are expressed by means of formal properties. These overcome the ambiguity of natural languages and are verified by means of either static (e.g., model checking) or, more frequently, dynamic (e.g., simulation) techniques. Therefore ABV provides a proof of correctness for the outcome of the MDD flow. Consequently, the MDD and ABV approaches have been combined to create efficient and effective design and verification frameworks that accompany designers and verification engineers throughout the system-level design flow of complex embedded systems, both for the Hardware (HW) and the Software (SW) parts (STM Products radCHECK, 2012. http://www.verificationsuite.com; Seger, Integrating design and verification – from simple idea to practical system. In: Proceedings of ACM/IEEE MEMOCODE, pp 161–162, 2006). It is, indeed, worth noting that to achieve a high degree of confidence, such frameworks require to be supported by *functional qualification* methodologies, which evaluate the quality of both the properties (Di Guglielmo et al. The role of mutation analysis for property qualification. In: 7th IEEE/ACM international conference on formal methods and models for co-design, MEMOCODE'09, pp 28–35, 2009. DOI 10.1109/MEMCOD.2009.5185375) and the testbenches which are adopted during the overall flow (Bombieri et al. Functional qualification of TLM verification. In: Design, automation test in Europe conference exhibition, DATE'09, pp 190–195, 2009. DOI 10.1109/DATE.2009.5090656). In this context, the goal of the chapter consists of providing, first, a general introduction to MDD and ABV concepts and related formalisms and then a more detailed view on the main challenges concerning the realization of an effective semiformal ABV environment through functional qualification.

Acronyms

ABV	Assertion-Based Verification
CTL	Computation Tree Logic
DUV	Design Under Verification
EFSM	Extended Finite-State Machine
ES	Embedded System
ESL	Electronic System Level
FSM	Finite-State Machine
HDL	Hardware Description Language
HW	Hardware
I/O	Input/Output
LLVM	Low-Level Virtual Machine
LTL	Linear Time Logic
MARTE	Modeling and Analysis of Real-Time Embedded Systems
MDA	Model-Driven Architecture

MDD	Model-Driven Design
MLBJ	Multi-Level Back Jumping
MMIO	Memory-Mapped I/O
MoC	Model of Computation
OMG	Object Management Group
OSCI	Open SystemC Initiative
OVL	Open Verification Library
PIM	Platform Independent Model
PSL	Property Specification Language
PSM	Platform Specific Model
RTL	Register Transfer Level
SERE	Sequential Extended Regular Expression
SoC	System-on-Chip
SVA	System Verilog Assertions
SW	Software
TLM	Transaction-Level Model
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Contents

22.1 Introduction to Model-Driven Design

The focus of MDD is to elevate the system development to a higher level of abstraction than that provided by HW description languages (e.g., VHDL and Verilog) for Hardware (HW) aspects and by third-generation programming languages for Software (SW) aspects [92]. The development is based on *models*, which are

abstract characterizations of requirements, behavior, and structure of the embedded system without anticipating the implementation technology.

Due to the noticeable effort of the Object Management Group (OMG) [84], the Unified Modeling Language (UML) [85] was originally adopted as the reference modeling language for describing software, and then it was also applied to the description of embedded hardware. UML provides general-purpose graphic elements to create visual models of systems and attempts to be generic in both the integration and the execution semantics. Due to such a general-purpose semantics, more specific *UML profiles* have been introduced for dealing with specific domains or concerns. They extend subsets of the UML meta-model with new standard elements, and they refine the core UML semantics to cope with particular hardware/software problems. For example, the SysML [97] profile supports the specification, the analysis, and the design of complex systems, which may include physical components. The Gaspard2 [57] profile, instead, supports the modeling of System-on-Chip (SoC). The synchronous reactive [35] profile provides a restrictive set of activity diagrams and sequence diagrams with a clear and semantically sound way of generating valid execution sequences. The Modeling and Analysis of Real-Time Embedded Systems (MARTE) [83] profile adds capabilities to UML for modeling and analysis of real-time and embedded systems. Its modeling concepts provide support for representing time and time-related mechanisms, the use of concurrent resources and other embedded systems characteristics (such as memory capacity and power consumption). The analysis concepts, instead, provide model annotations for dealing with system property analysis, such as schedulability analysis and performance analysis. It is worth noticing that other UML profiles exist for hardware-related aspects such as system-level modeling and simulation [81,90]. Other hardware-oriented profiles and a comparison of them are clearly described in [18]. Model-Driven Design (MDD) has been also used for modeling embedded systems that interact together through communication channels to build distributed applications. In this context, the MDD approach consists in using a *UML deployment diagram* to capture the structural representation of the whole distributed application. MARTE stereotypes (e.g., the MARTE-GQAM sub-profile and MARTE nonfunctional properties) can be used to represent attributes such as throughput, price, and power consumption. Furthermore some aspects (e.g., node mobility) require the definition of an ad-hoc UML network profile [46].

Besides the standard UML supported by OMG, some proprietary variants of the UML notations also exist. The most famous ones are the MathWorks' Stateflow and Simulink [98] formalisms. They use finite-state, machine-like and functional-block, diagram-like models, respectively, for specifying behavior and structure of reactive hardware/software systems with the aim of rapid embedded SW prototyping and engineering analysis. Several MDD tools on the market support UML and Model-Driven Architecture (MDA). The underlying idea of MDA is the definition of models at different levels of abstraction which are linked together to form an implementation. MDA distinguishes the conceptual aspects of an application from their representation on specific implementation technologies. For this reason, the MDA design approach uses Platform Independent Models (PIMs) to specify

what an application does, and Platform Specific Models (PSMs) to specify how the application is implemented and executed on the target technology. The key element of an MDA approach is the capability to automatically transform models: transformation of PIM into PSM enables realizations, whereas transformations between PIMs enable integration features.

22.2 Introduction to Assertion-Based Verification

Assertion-based verification aims at providing verification engineers with a way to formally capture the intended specifications and checking their compliance with the implemented embedded system. Specifications are expressed by means of formal properties defined according to temporal logics, e.g., Linear Time Logic (LTL) or Computation Tree Logic (CTL), and expressed by means of assertion languages, like Property Specification Language (PSL) [64].

Approaches based on ABV are traditionally classified in two main categories: *static* (i.e., *formal*) and *dynamic* (i.e., *simulation based*). In static Assertion-Based Verification (ABV), formal properties, representing design specifications, are exhaustively checked against a formal model of the design by exploiting, for example, a model checker. Such an exhaustive reasoning provides verification engineers with high confidence in system reliability. However, the well-known state space explosion problem limits the applicability of static ABV to small/medium-size, high-budget, and safety-critical projects [67]. On the contrary, thanks to the scalability provided by simulation-based techniques, dynamic ABV approaches are nowadays preferred for verifying large designs, which have both reliability requirements and stringent development cost/time-to-market constraints. In particular, in the hardware domain, dynamic ABV is affirming as a leading strategy in industry to guarantee fast and high-quality verification of hardware components [24, 80], and several verification approaches have been proposed [50, 51]. In dynamic ABV, properties are compiled into *checkers* [17], i.e., modules that capture the behavior of the corresponding properties and monitor if they hold with respect to the design, when the latter is simulated by using a set of (automatically generated) stimuli.

22.3 Integrating MDD and ABV

Even if MDD simplifies software implementation, it does not prevent the designer from wrongly defining system behavior. Certain aspects concerning the verification of the code generated by MDD flows are automated, as, for example, the structural analysis of code, but specification conformance, i.e., functional verification, is still a human-based process [47,55]. Indeed, the de facto approach to guarantee the correct behavior of the design implementation is monitoring system simulation: company verification teams are responsible for putting the system into appropriate states by generating the required stimuli, judging when stimuli should be executed, manually simulating environment and user interactions, and analyzing the results to identify

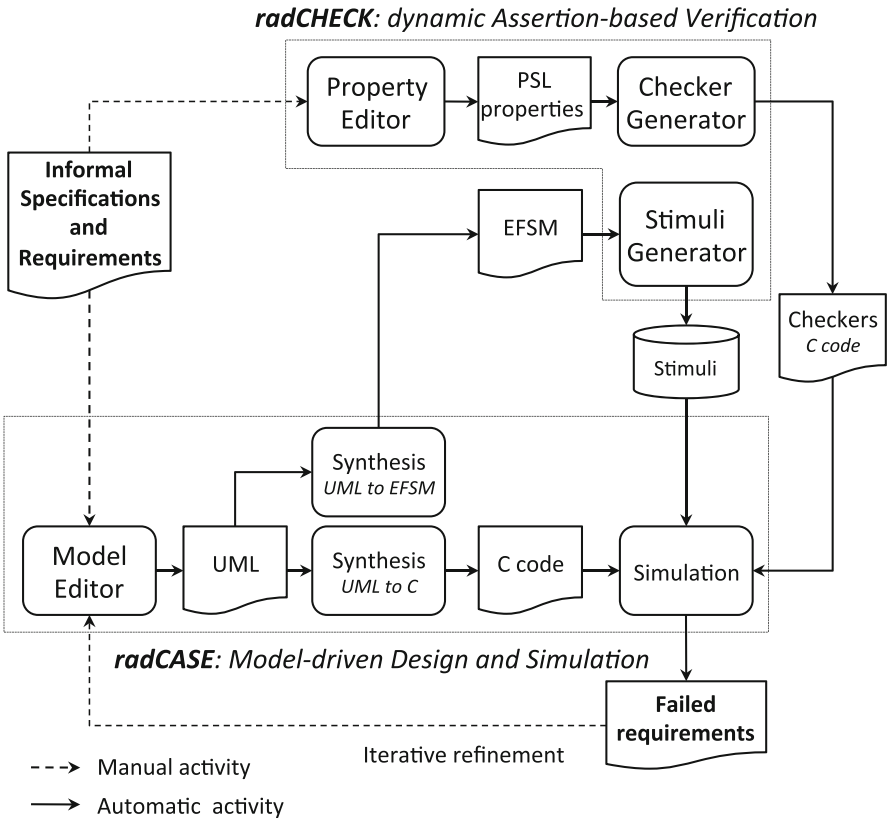


Fig. 22.1 The combined model-driven and verification framework implemented in the RadSuite

unexpected behaviors. MDD and dynamic ABV, if combined in a comprehensive framework, enable automatic functional verification of both embedded SW and HW components. The approach generally relies on a design and verification framework composed of two environments: a UML-like modeling and development environment - supporting model-driven design - and a dynamic ABV environment that supports assertion definition and automatic checkers and stimuli generation.

An example of this design and verification flow, which integrates MDD and ABV, is represented by the commercial RadSuite, composed by radCASE and radCHECK [38] (Fig. 22.1). Starting from informal specifications and requirements, the designer, with the model editor of radCASE, defines the system model by using a UML-based approach. Concurrently, with the property editor of radCHECK, he/she defines a set of PSL assertions that the system must fulfill. Then, radCASE automatically translates the UML specifications in the C-code implementation, and it automatically extracts an Extended Finite-State Machine (EFSM) model (see Sect. 22.4.1.1 for this formalism) to support automatic verification. At the same time, radCHECK can be used to automatically generate executable checkers from the defined PSL assertions. The dynamic ABV is guided by stimuli automatically

generated by a corner-case-oriented concolic stimuli generator that exploits the EFSM model to explore the system state space (see Sect. 22.7). Checkers execute concurrently with the Design Under Verification (DUV) and monitor if it causes any failure of the corresponding properties. The designer uses the resulting information, i.e., failed requirements, for refining the UML specifications incrementally and in an iterative fashion.

22.4 Models and Flows for Verification

The key ingredient underpinning an effective design and verification framework based on MDD and ABV is represented by the possibility of defining a model of the desired system and then automatically deriving the corresponding simulatable description to be used for virtual prototyping. Selecting the formalism to represent the model is far from being a trivial choice, as the increasing complexity and heterogeneity of embedded systems generated, over time, a plethora of languages and different representations, each focusing on a specific subset of the Embedded System (ES) and on a specific domain [44]. Examples are EFSMs, dedicated to digital HW components and cycle-accurate protocols, hybrid automata for continuous physical dynamics, high-level UML diagrams for high-level modeling of hardware, software, and network models. This heterogeneity reflects on the difficulty of standardizing automatic approaches that allow the conversion of the high-level models in executable specifications (e.g., SystemC/C code). Such an automatic conversion is indeed fundamental to reduce verification costs, particularly in the context of virtual prototyping of complex systems that generally integrate heterogeneous components through both bottom-up and top-down composition flows. In this direction, automata-based formalisms represent the most suitable solutions to enable a precise mapping of the model into simulatable descriptions. Thus, the following discussion in this section is intended to summarize the main automata-based formalisms available as state of the art, together with bottom-up and top-down flows, whose combined adoption allows the generation of a homogeneous simulatable description of the overall system.

22.4.1 Automata-Based Formalisms

The most widespread models for representing the behavior of a component or a system are based on automata, i.e., models that rely on the notions of states and of transitions between states. The simplest automata-based model, i.e., the finite-state machine, has proved to be too strict to allow a flexible and effective view of modern components. This originated a number of extensions, each targeting specific aspects and domains, as summarized in the following of this section.

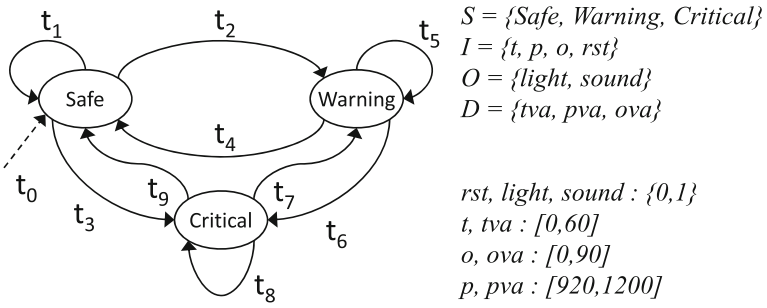
22.4.1.1 Extended Finite-State Machines

EFSMs extend standard Finite-State Machines (FSMs) to allow a more compact representation of the system that reduces the risk of state explosion for complex designs [4]. An EFSM differs from the traditional FSM, since transitions between

states are not labeled in the classical form input/output values, but they take care of the values of internal variables too. Practically, transitions are labeled with an *enabling function*, which represents the guard of the transition, and an *update function*, which specifies how the values of variables and outputs evolve when the transition is fired upon the satisfaction of its guard.

To exemplify the concept, Fig. 22.2 reports the EFSM of a simplified in-flight safety system. The states of the EFSM are $S = \{Safe, Warning, Critical\}$, where $Safe$ is the reset state. The input variables are $I = \{t, p, o, rst\}$ and represent the corresponding temperature, pressure, and oxygen variables, whereas rst represents the EFSM reset signal. $O = \{light, sound\}$ is the set of output variables corresponding to light and sound controls. Finally, $D = \{tva, pva, ova\}$ is the set of internal variables of the EFSM. For each transition, the enabling function and update function are reported in the table under the figure. For readability, only a reset transition t_0 is depicted with a dotted arrow as a representative of each of the reset transitions outgoing from the states of the EFSM and entering in *Safe*.

Unfortunately, the semantics of EFSMs is strictly discrete, and it does not support continuous-time physical models. Thus it cannot, for instance, represent an ES with its continuous-time environment.



Transition	Enabling Function	Update Function
t_0	$rst = 1$	$tva = 0;\ ova = 0;\ pva = 0;$ $light = 0;\ sound = 0;$
t_1	$t < 42 \wedge p \geq 980$	–
t_2	$t \geq 42$	$tva = t;\ pva = p;\ light = 1;$
t_3	$t < 42 \wedge p > 1020$	$tva = t;\ pva = p;\ sound = 1;$
t_4	$t < 42 \wedge p \leq 1020$	$light = 0;\ sound = 0;$
t_5	$t \geq 42 \wedge p > 1020 \wedge$ $t \leq tva \wedge o \geq 18$	$tva = t;$
t_6	$t \geq tva \wedge o < 18$	$ova = o;\ sound = 1;$
t_7	$o > ova \wedge p < pva$	$sound = 0;$
t_8	$o \leq ova \wedge p \geq pva \wedge$ $t \geq 42 \wedge p > 1020$	–
t_9	$t < 42 \wedge p \leq 1020$	$light = 0;\ sound = 0;$

Fig. 22.2 An EFSM specification of a simplified in-flight safety system

22.4.1.2 Hybrid Automata

A hybrid automaton is modeled as a set of states and transitions, but, as opposed to EFSMs, it supports both discrete time and continuous time dynamics [58]. The discrete time dynamics coincides with FSM semantics, and it is implemented through transitions between states that respond to system evolution and to synchronization events. The continuous-time dynamics is implemented by the states, which are associated with two predicates: the *flow* predicate that constrains the evolution of continuous variables into the state, and the *invariant* predicate, that specifies whether it is possible to remain into the state or not, depending on a set of conditions on variables. Variables can be assigned continuous values, as opposed to EFSMs.

Figure 22.3 depicts an example of a simple hybrid automaton. If compared with Fig. 22.2, the automaton now associates each state with an invariant condition (e.g., $x \geq 18$, for the state $S0$) and with a flow predicate which shows the rate of change of the variable x with time (e.g., $x' = -0.1x$, for the state $S0$, where x' represents the first derivative of variable x). Furthermore, a synchronization event, $close$, is used to force the transition to $S1$, irrespective of the current state of variable x.

A special class of automata, namely, *timed automata*, introduces the notion of time [6]. Time evolution is modeled with dense variables, called clocks, whose evolution is constrained by predicates and used in transition guards. Furthermore, events and variables are associated with a time stamp. As a result, automaton evolution depends also on time. This is extremely useful in the modeling of real-time systems with time constraints.

Hybrid automata are especially suited for modeling control scenarios, modeling a tight interaction between a controller and a continuous time environment. However, this formalism is suited for modeling only high-level systems. Unfortunately, describing cycle-accurate hardware behaviors as well as software functionalities (such as interrupt handling) would exponentially increase the complexity of the related model, and it would lead to state space explosion [56].

22.4.1.3 UML Diagrams

UML is a standardized general-purpose modeling language, specially suited for modeling SW intensive systems, but often adopted also for modeling HW components and networked systems [85]. UML includes a set of graphic notation techniques for clearly representing different aspects of a system, i.e., its structure (*structural diagrams*) or its behavior (*behavioral diagrams*).

The most useful class from the point of view of MDD approaches for HW/SW systems is represented by *behavioral diagrams*, which model what happens in the system, either in terms of internal behavior or from the communication perspective.

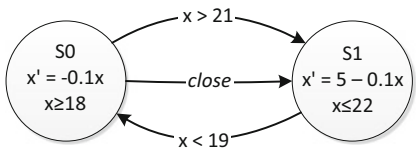


Fig. 22.3 Example of a hybrid automaton

Behavioral diagrams can be further classified into different classes, among which the most relevant for modeling HW/SW systems are:

- *Activity diagrams*, which represent the data and control flow between activities
- *Interaction diagrams*, which represent the interaction between collaborating parts of a system, in terms of message exchange (*sequence diagram* and *communication diagrams*), and of state evolution depending on timed events (*timing diagrams*)
- *State machine diagrams*, which are automata-based model representing the state transitions and actions performed by the system in response to events

Besides behavioral diagrams, *deployment diagrams*, belonging to the category of structural diagrams, have been also used in the context of MDD to capture the structural representation of network aspects, in conjunction with MARTE stereotypes [46].

In general, UML diagrams have been specialized to fit many different domains through the definition of profiles. However, their diagrams are too high level to represent cycle accurate models and physical models with a sufficient accuracy, without incurring in the state explosion problem or degenerating into standard FSMs.

Figure 22.4 shows a sequence diagram example. Sequence diagrams show how processes operate and their interactions, represented as exchanged messages. For this reason, they are the most common diagrams to specify system functionality, communication, and timing constraints. Lifelines (the vertical dashed lines) are the objects constituting the system. The rectangles in a lifeline represent the execution of a unit of behavior or of an action, and they are called execution specification. Execution specifications may be associated with timing constraints that represent either a time value (③) or a time range (④). Finally, messages, written with horizontal arrows, display interaction. Solid arrowheads represent synchronous calls, open arrowheads represent asynchronous messages, and dashed lines represent reply messages.

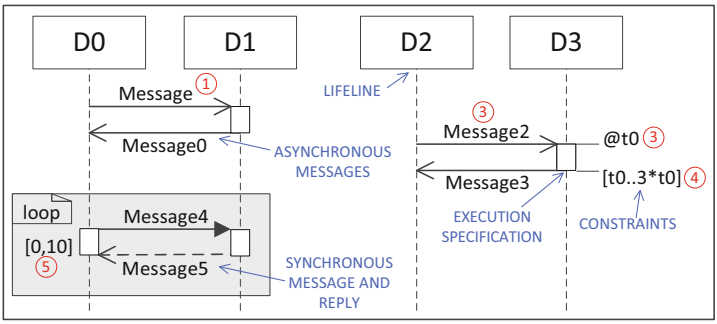


Fig. 22.4 Example of UML sequence diagram

22.4.1.4 The UNIVERCM Model of Computation (MoC)

UNIVERCM is an automaton-based MoC that unifies the modeling of both the analog (i.e., continuous) and the digital (i.e., discrete) domains, as well as hardware-dependent SW. A formal and complete definition is available in [44, 56].

In each UNIVERCM automaton (depicted in Fig. 22.5), states reproduce the continuous-time dynamics of hybrid automata, while transitions reproduce the discrete-time semantics of EFSMs. As a consequence, UNIVERCM automata can be reduced to either EFSMs or hybrid automata, depending on the enabled features. An automaton can be transformed in an *equivalent EFSM* by transforming its continuous time features into discrete transitions (e.g., by discretizing the flow predicate). A UNIVERCM automaton can also be transformed into an equivalent hybrid automaton by reducing discrete transitions to conditions that allow to change state and by moving the corresponding actions to the flow predicate of the destination state. This correspondence of UNIVERCM automata to well-established formal models allows to apply well-known design or verification techniques, originally defined for EFSMs or hybrid automata, also to the context of UNIVERCM. This makes UNIVERCM an important resource in the design of HW/SW systems, as it covers the heterogeneity of ES, ranging from analogue and digital HW to dedicated SW, in order to build reuse and redesign flows [44].

Note that UNIVERCM states and transitions are provided with two additional tags, i.e., atomic and priority. The *atomic* tag is used to define atomic regions that are considered to all intent a single transition when performing parallel composition with other automata. The *priority* tag is used to handle nondeterministic behaviors that may be present in a system: in case two or more transitions can be performed at the same time, the automaton activates the one with lower value of the priority tag.

UNIVERCM variables fall back in three main classes: discrete variables, wire variables, and continuous variables. Wire variables extend discrete variables with an event that is activated whenever the corresponding variable changes value. This mechanism is used to mimic the event-driven semantics of Hardware Description Languages (HDLs). Continuous variables are dense variables that can be either assigned an explicit value (e.g., in discrete transitions) or constrained through the flow predicate. They thus resemble the clock variables of timed automata [6].

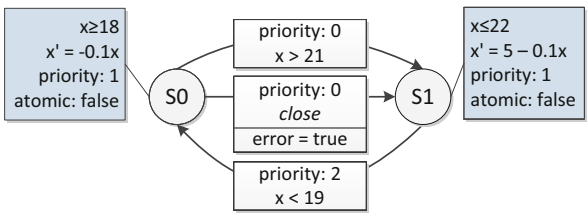


Fig. 22.5 Example of UNIVERCM automaton

22.4.2 Top-Down and Bottom-Up Flows for System Verification

UNIVERCM bridges the characteristics of the major automata-based formalisms, and it thus allows to reduce both top-down and bottom-up flows to a single framework, to build a homogeneous simulatable description of the overall ES. Such a description can then be the focus of redesign and validation flows, targeting the homogeneous simulation of the overall system [43]. This section presents the main flows that can be reduced to UNIVERCM, as summarized in Fig. 22.6.

22.4.2.1 Bottom-Up: Mapping Digital HW to UNIVERCM

The mapping of digital HW to UNIVERCM can be defined focusing on the semantics of HDLs, i.e., of the languages used for reproducing and designing HW execution. In HDLs, digital HW is designed as a number of concurrent processes that are activated by events and by variations of input signals, constituting the process sensitivity list. Process activation is managed by an internal scheduler that repeatedly builds the queue of runnable processes and advances simulation time.

When mapping to UNIVERCM, each HDL process is mapped to an automaton, whose transitions are activated by variations in the value of the signals in the sensitivity list. In the example in Fig. 22.7, the automaton is activated by events on the read signal `b` that fires the transaction from state `H0` to state `H1`. This mechanism, together with the sharing of variables and signals, ensures that process communication and interaction are correctly preserved. Note that, since digital HW does not foresee continuous-time evolution, the automaton is restricted to the discrete-time dynamics (i.e., to an EFSM). The existence of predefined synchronization points (e.g., wait primitives) is ensured in UNIVERCM with an ad hoc predicate, called *atomic*, that allows to consider a number of transitions and states as a single transition (e.g., transitions from `H0` to `H4` in Fig. 22.7). This guarantees that the original execution flow is preserved.

The mapping of the sensitivity list to events, and the parallel semantics of UNIVERCM automata, builds an automatic scheduling mechanism that avoids the need for an explicit scheduling routine. The advancement of simulation time is explicitly modeled with an additional automaton [43].

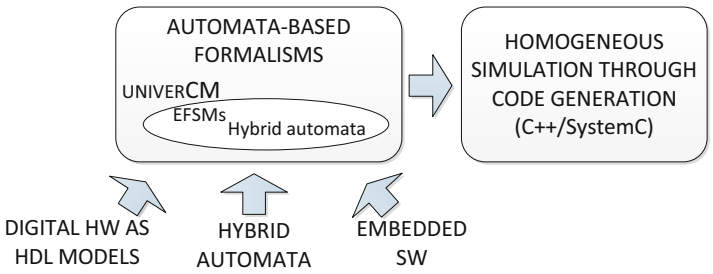


Fig. 22.6 Top-down and bottom-up flows proposed in the following of this section

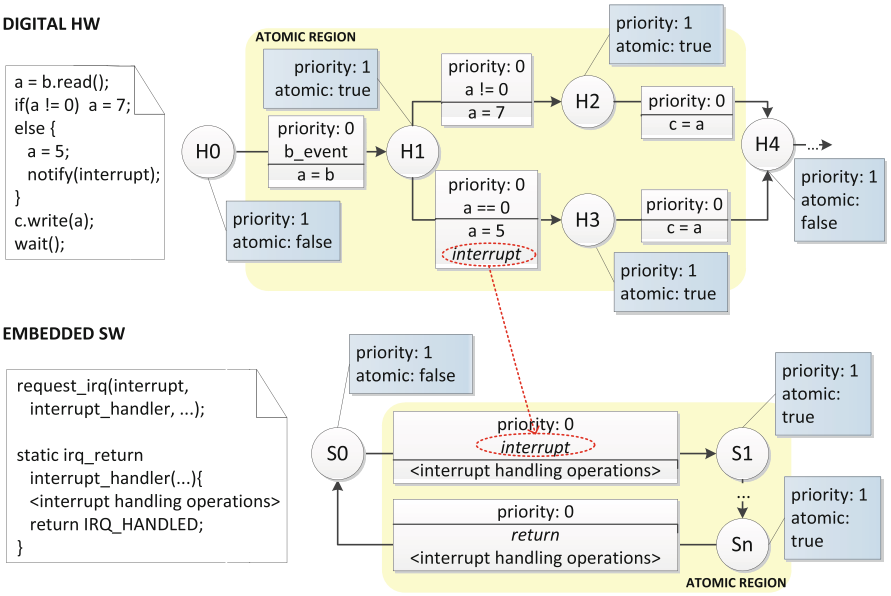


Fig. 22.7 Mapping of a digital HW component and of the communicating embedded SW to UNIVERCM automata

22.4.2.2 Bottom-Up: Mapping Embedded SW to UNIVERCM

Embedded SW is typically structured into a number of functions that can be easily represented as UNIVERCM automata evolving among a certain set of states via transitions. Since SW does not allow continuous evolution, each automaton is restricted to its discrete-time dynamics (i.e., to an EFSM).

Each function is provided with an activation event (for representing function invocation, event *interrupt* in Fig. 22.7) and a return label, which is used to communicate to the caller that the function has finished its execution (event *return* in Fig. 22.7).

Note that the atomic predicate can be used also in case of SW to avoid race conditions and unpredictable behaviors due to concurrency, e.g., in Fig. 22.7 all transitions are encapsulated in an atomic region, to guarantee that the execution of function `interrupt_handler` is non-interruptible.

Communication with HW devices based on the Memory-Mapped I/O (MMIO) approach is easily implemented in UNIVERCM by representing MMIO locations as variables shared with the HW automata. HW interrupts are mapped to synchronization events. The interrupt handling routine is mapped to an automaton, just like any other function. The activation event of the automaton is the interrupt fired by the HW device. The automaton remains suspended until it receives the interrupt event and, on receipt of the event, it executes the necessary interrupt handling operations. In the example of Fig. 22.7, the activation event of the function (event *interrupt*) is the interrupt fired by the HW automaton (in the transition from `H1` to `H3`, as highlighted by the red arrow).

22.4.2.3 Bottom-Up: Mapping Hybrid Automata to UNIVERCM

Since UNIVERCM is a superset of hybrid automata, this mapping is quite straightforward. Care must be taken in the mapping of synchronization events, as hybrid automata may activate an event only if all its recipients may perform a transition in response to the event. To reproduce this semantics in UNIVERCM, each recipient automaton is provided with a flag variable that is false by default and that is set to true only if the current state has an outgoing transition fired by the synchronization event. The event may be fired only after checking that all the corresponding flag variables are true.

Hybrid automata may be hierarchical, for simplifying the design of analog components. Mapping a hierarchical hybrid automaton to UNIVERCM requires to remove the hierarchy by recursively flattening the description.

22.4.2.4 Top-Down: Mapping UML Diagrams to UNIVERCM

The mapping of UML diagrams is defined to EFSMs, that are the reference automata-based model for MDD approaches. Given that EFSMs can be considered the discrete-time subset of UNIVERCM, this is equivalent to mapping the diagrams to UNIVERCM automata. For this reason, in the following, the terms EFSM and UNIVERCM automaton are interchangeable.

The mapping is defined for *UML sequence diagrams*. These are the most common diagrams to specify system functionality, communication, and timing constraints. However, a similar approach can be applied also to other classes of UML diagrams.

The mapping of sequence diagrams to UNIVERCM is exemplified in Fig. 22.8. Each diagram is turned into one automaton, whose states are defined one per lifeline (*D0*, *D1*, *D2*, and *D3*) [45]. Message receipt forces transition from one state to the next (e.g., from *D0* to *D1* at ①). Messages are enumerated, to define enabling conditions that preserve the execution order imposed by the sequence diagram (②). This allows to reproduce also control constructs, such as the loop construct in Fig. 22.8, that iterates a message transfer ten times (⑤). Timing constraints are used to perform check constraints through an additional state *W* (③ and ④) that may raise timing errors.

22.4.2.5 Top-Down: Mapping UNIVERCM Automata to C++/SystemC

UNIVERCM has been specifically defined to ease the conversion of UNIVERCM automata toward C++ and SystemC descriptions [43, 44].

Each UNIVERCM automaton is mapped in a straightforward manner to a C++ function containing a *switch* statement, where each case represents one of the automaton states. Each state case lists the implementation of all the outgoing edges and of the delay transition provided for the state. When an edge is traversed, variables are updated according to the update functions, and the continuous flow predicates and synchronization events are raised, where required.

The code generated from UNIVERCM automata is ruled by a *management function*, in charge of activating automata and of managing the status of the overall system and parallel composition of automata.

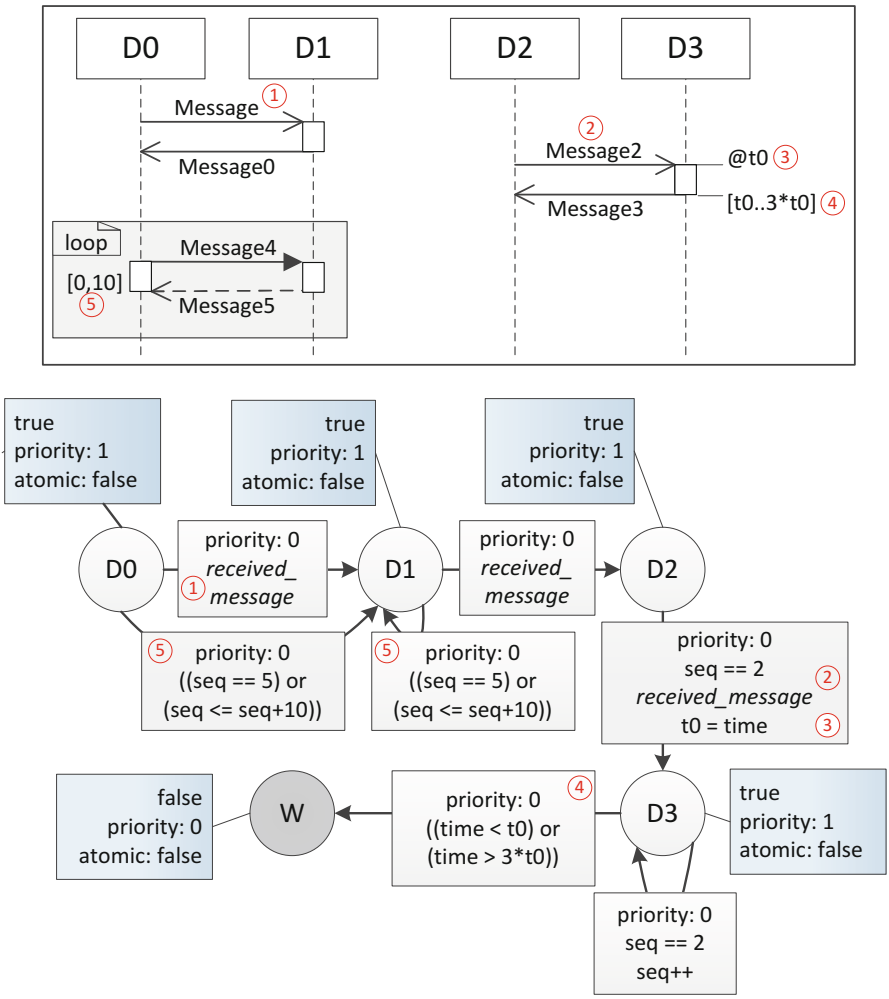


Fig. 22.8 Mapping of a UML sequence diagram to an EFSM (and thus to the discrete-time restriction of UNIVERCM)

In case of conversion toward SystemC descriptions, the presence of a simulation kernel allows to delegate some management tasks and to reproduce automata behavior through native SystemC constructs. Thus, UNIVERCM automata are mapped to processes, rather than functions. This allows to delegate automata activation to the SystemC scheduler, by making each process sensitive to its input variables. Automata activation is removed from the management function, that still updates the status of variables and events at any simulation cycle. The management function itself is declared as a process.

22.5 Assertion Definition and Checker Generation

In software verification, software designers widely use *executable assertions* [59] for specifying conditions that apply to some states of a computation, e.g., "pre-conditions" and "post-conditions" of a procedural code block. A runtime error is released whenever execution reaches the location at which the executable assertion occurs and the related condition does not hold any more. This kind of executable assertions is limited to Boolean expressions, which are totally unaware of temporal aspects. However, if the designers aim to check more complex requirements in which Boolean expressions are used for defining relations spanning over the time, they have to (i) define assertions in a formal language and (ii) synthesize them as executable modules, i.e., *checkers*. Checkers, integrated into the simulation environment, monitor the software execution for identifying violations of the intended requirement.

In hardware verification, several solutions have already been proposed. These approaches can be classified in (i) library based or (ii) language based.

Library-based approaches rely on libraries of predefined checkers, e.g., Open Verification Library (OVL) [52], which can be instantiated into the simulation environment for simplifying the checking of specific temporal behaviors. Unfortunately, due to their inflexibility for checking general situations, the predefined checkers limit the completeness of the verification.

Language-based approaches, instead, use declarative languages, such as PSL [64] and System Verilog Assertions (SVA) [94], for formalizing the temporal behaviors into well-defined mathematical formulas (i.e., assertions) that can be synthesized into executable checkers by using automatic tools named checker generators [2, 16, 17, 34]. These tools may generate checkers implementations at different levels of abstraction, from the Register Transfer Level (RTL), e.g., MBAC [17] and FoCs [2], to the C-based Electronic System Level (ESL) [33], e.g., FoCs.

However, a large part of an ES is software, which must also be verified. Some attempts have been tried to extend hardware ABV to embedded software and a comprehensive work is in [38].

In [26], the authors present a Microsoft's proprietary approach for binding C language with PSL. They define a subset of PSL and use a simulator as an execution platform. In this case, only a relatively small set of temporal assertions can be defined, since only the equality operator is supported for Boolean expressions, and the simulator limits the type of embedded software applications.

Another extension of PSL is proposed in [101], where the authors unify assertion definition for hardware and software by translating their semantics to a common formal semantic basis. In [100], the authors use temporal expressions of the *e* hardware verification language to define checkers. In both these cases, temporal expressions are similar but not compatible with PSL standards.

Finally, in [74] the authors propose two approaches based on SystemC checkers. In the first case, embedded software is executed on top of an emulated SystemC

processor, and, at every clock cycle, the checkers monitor the variables and functions stored in the memory model. In the second approach, embedded software is translated to SystemC modules which run against checkers. In this case, timing reference is imposed by introducing an event notified after each statement, and the SystemC process is suspended on additional `wait()` statements. In both cases, there are several limitations. First, the approaches are not general enough to support real-life embedded software: the SystemC processor cannot reasonably emulate real embedded system processors. In addition, the translation of embedded software applications in SystemC may be not flexible enough. Secondly, in both cases the SystemC cosimulation and the chosen timing references introduce significant overhead. In particular, clock cycle or statement step may be an excessively fine granularity for efficiently evaluating a sufficient number of temporal assertions. Moreover, on the one hand, defining assertions which consider absolute time may generate significantly large checkers to address the high number of intermediate steps; on the other, it is difficult to define temporal assertions at source code level, i.e., C applications, in terms of clock cycles.

22.5.1 Template-Based Assertion Design

Previous sections show that assertion definitions can be unified for both hardware and software, that is, they can be applied to an entire ES. An effective tool in this case is DDPSL [41]: a template library and a tool which simplify the definition of formal properties. It combines the advantages of both PSL and OVL, i.e., expressiveness and simplicity.

The template library is composed of *DDTemplates*, i.e., PSL-based templates accompanied with an *interface semantics*. The adoption of a PSL-based property implementation guarantees the same expressiveness of LTL and CTL temporal logics, a wide compatibility with HDL (e.g., VHDL, Verilog and SystemVerilog, SystemC) and programming languages (e.g., C++), and enables a large reuse of already available verification tools previously described. Moreover, like the OVL approach, the use of an interface semantics allows a clean separation between property implementation and property semantics. Such an interface notably simplifies property definition: the user needs only to understand the semantics of the interface and replace parameters with the intended expression, and a correct-by-construction PSL code is automatically generated.

DDTemplates are characterized by (i) a parametric interface, (ii) a formal PSL implementation, and (iii) a detailed semantics (i.e., interface semantics) that specifies how to use the corresponding interface for defining properties.

The interface consists of a synthetic description that gives an intuitive idea of the semantics of the property. Such a description is characterized by parameters that are placeholders inside the property. These parameters are strongly typed and distinguished into Boolean, arithmetic, temporal, and Sequential Extended Regular Expression (SERE) parameters. The interaction with such placeholders guides

property definition: the user can replace parameters only with compatible elements according to a predefined semantics check.

Figure 22.9 shows an example of the "conditional events bounded by time" DDTemplate. In particular, it reports the synthetic description adopted as interface showing the three parameters (i.e., $\$P$, $\$Q$, and $\$i$) that the user can substitute.

Although the interface is explanatory, major details of the interface semantics are described by means of online documentation provided by the DDEditor. Such documentation reports information related to the parameters type and their meaning, the temporal behavior the template aims to check and possible warnings. For example, the online documentation corresponding to the DDTemplate shown in Fig. 22.9 reports:

- *Semantics of the parameters*
  - **$P** is a *Boolean expression* that represents a configuration, an event or an input/output for the system/program.
  - **$i** is an *integer* that specifies the instant in the future within which **$Q** must hold.
  - **$Q** is a *Boolean expression* that represents a new configuration, an event or an input/output for the system/program.
- *Semantics of the template*
  - The template specifies that if the system takes the configuration $\$P$ (or the event $\$P$ happens) in the cycle t_0, then the new configuration or the event $\$Q$ must occur within the cycle $t_0 + \$i$.
- *Warnings*
  - Notice that $\$Q$ may occur in many cycles within $t_0 + \$i$, but it is not possible that $\$Q$ never happens within the cycle $t_0 + \$i$.

The PSL implementation, instead, consists of a formal PSL definition (Fig. 22.10).

More than 60 templates have been defined and organized into five libraries (e.g., a selection of templates is reported in Table 22.1); each one focuses on a specific category of patterns: universality, existence, absence, responsiveness, and precedence. The *universality* library describes behaviors that must hold continuously during the software execution (e.g., a condition that must be preserved for the whole execution, or that has to hold continuously after that the software reaches a particular configuration). The *existence* library describes behaviors in which the occurrence of particular conditions is mandatory for the software execution (e.g., a condition must be observed at least once during the whole

Every time that **$P** then, within **$i** cycle(s), **$Q**

Fig. 22.9 Example of a DDTemplate interface

always (**%P** → next_e[0..**%i**](**%Q**))

Fig. 22.10 Example of formal PSL definition

Table 22.1 Selection of assertion templates

Library	Parametric interface	Parametric PSL definition
Universality	P holds continuously after Q	`always ($Q -> next (always $P))`
	P holds continuously since Q up to R	`always (($Q & !$R) -> ($P until_ $R))`
Existence	P holds at least once since Q	`next_event!($Q)(eventually! $P)`
	P holds at least once since Q up to R	`always (($Q & !$R) -> (!$R until! $P))`
Absence	P is continuously false after Q until R	`always (($Q & !$R) -> next (!$P until! $R))`
	P is continually false before R	`!$P until! $R`
Responsiveness	P causes S to happen	`always ($P -> eventually! ($S))`
	P causes S to happen, but after Q	`always ($P -> (($Q before! $S) & eventually! ($S)))`
Precedence	P precedes R globally	`always ($P -> ($P before $R))`
	P precedes R before S	`always ($P ->(($P before $R)&($R before! $S)& eventually! ($S)))`

execution or after that a particular configuration is reached, etc.). The *absence* library describes behaviors that must not occur during the software execution or under certain conditions. The *responsiveness* library, instead, describes behaviors that specify cause-effect relations (e.g., a particular condition implies a particular configuration of the software variables). Finally, the *precedence* library describes behaviors that require a precise ordering between conditions during the software execution (e.g., a variable has to assume specific values in an exact order).

Notice that the user can ignore the exact PSL formalization. He/she can define properties by simply dragging and dropping expressions onto placeholders contained into the interface by exploiting the DDEditor tool [41].

22.6 Mutant-Based Quality Evaluation

Assertion-based verification can hypothetically provide an exhaustive answer to the problem of design correctness, but from the practical point of view, this is possible only if (1) the DUV is stimulated with testbenches that generate the set of all possible input stimuli and (2) a complete set of formal properties is defined that totally captures the designer's intents. Unfortunately, these conditions represent two of the most challenging aspects of dynamic verification, since the set of input stimuli for sequential circuits is generally infinite, and the answer to the question "have I written enough properties?" is generally based on the expertise of the verification engineers. For these reasons, several metrics and approaches have been defined to address the **functional qualification** of dynamic verification methodologies and frameworks, i.e., the evaluation of the effectiveness of testbenches and properties adopted to check the correctness of a design through semiformal simulation-based techniques.

Among existing approaches, mutation analysis and mutation testing [37], originally adopted in the field of software testing, have definitely gained consensus, during the last decades, as being important techniques for the functional qualification of complex systems both in their software [61] and hardware [15] components.

Mutation analysis [86] relies on the perturbation of the DUV by introducing syntactically correct functional changes that affect the DUV statements in small ways. As a consequence, many versions of the model are created, each containing one mutation and representing a *mutant* of the original DUV. The purpose of such mutants consists in perturbing the behavior of the DUV to see if the test suite (including testbenches and, in case, also properties) is able to detect the difference between the original model and the mutated versions. When the effect of a mutant is not observed on the outputs of the DUV, it is said to be undetected. The presence of undetected mutants points out inadequacies either in the testbenches, which, for example, are not able to effectively activate and propagate the effect of the mutant, or in the DUV model, which could include redundant code that can never be executed. Thus, mutation analysis has been primarily used for helping the verification engineers in developing effective testbenches to activate all DUV behaviors and discovering design errors. More recently, it has been used also to measure the quality of formal properties that are defined in the context of assertion-based verification. The next sections will summarize some of the most recent approaches based on mutation analysis for the functional qualification of testbenches and properties.

22.6.1 Testbench Qualification

Nowadays, (i) the close integration between HW and SW parts in modern embedded systems, (ii) the development of high-level languages suited for modeling both HW and SW (like SystemC with the Transaction-Level Model (TLM) library), and (iii) the need of developing verification strategies to be applied early in the design flow require the definition of simulation frameworks that work at the system level. Consequently, mutation analysis-based strategies for the qualification of testbenches need to be defined at system level too, possibly before HW and SW functionalities are partitioned. In this context, the mutation analysis techniques proposed for over 30 years in the SW testing community can be reused for perturbing the internal functionality of the DUV, which is indeed implemented like a SW program, often by means of C/C++ behavioral algorithms. In particular, several approaches [5, 12, 13, 20, 88, 89], empirical studies [75], and frameworks [19, 31, 36, 76] have been presented in the literature for mutation analysis of SW program. Different aspects concerning software implementation are analyzed in all these works, in which the approaches are mainly suited for perturbing Java or C constructs. However, all these proposals are suited to target basic constructs and low-level synchronization primitives rather than high-level primitives typically used for modeling TLM communication protocols. Other approaches present mutation operators targeting formal abstract models, independently from specific programming languages [9, 12, 88, 89].

These approaches are valuable to be applied at TLM levels. However, the authors do not show a strict relation between the modeled mutants and the typical design errors introduced during the modeling steps. To overcome this issue, a native TLM mutation model to evaluate the quality of TLM testbenches has been proposed in [14, 15]. It exploits traditional SW testing framework for perturbing the DUV functional part, but it presents a new mutation model for addressing the system-level communication protocol typical of TLM descriptions. The approach is summarized in the rest of this section.

22.6.1.1 Mutant-Based Qualification of TLM Testbenches

The approach assumes that the functionality of the TLM model is a procedural style of code in one or more SystemC processes. Therefore, the mutation model for the functionality is derived from the work in [36] that defined mutation operators for the C language. Selective mutation (suggested in [78] and evaluated in [87]) is applied to ensure the number of mutations grows linearly with the code size. On the contrary, the communication part of the DUV is mutated by considering the effect of perturbations injected on the EFSM models representing the Open SystemC Initiative (OSCI) SystemC TLM-2.0 standard primitives adopted for implementing blocking and non-blocking transaction-level interfaces.

In TLM-2.0, communication is generally accomplished by exchanging packets containing data and control values, through a channel (e.g., a socket) between an initiator module (master) and a target module (slave). For the sake of simplicity and lack of space, we report in Fig. 22.11 the EFSMs representing the primitives and the proposed mutations of the most relevant interfaces (i.e., blocking and non-blocking interfaces).

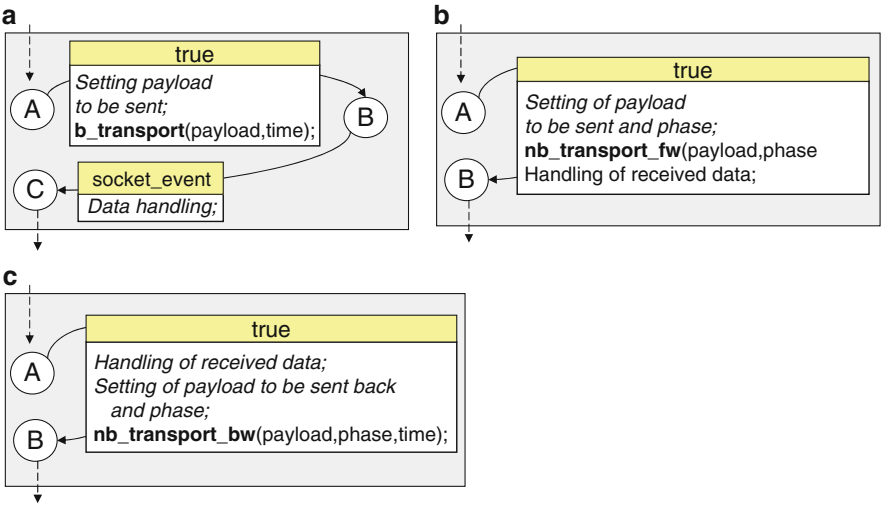


Fig. 22.11 EFSM models of some SystemC TLM-2.0 primitives

- *Blocking interface.* It allows a simplified coding style for models that complete a transaction in a single-function call, by exploiting the blocking primitive `b_transport()`. The EFSM model of primitive `b_transport()` is composed of three states (see Fig. 22.11a). Once the initiator has called `b_transport()`, the EFSM moves from state *A* (initial state) to state *B*, and it asks the socket channel to provide a payload packet to the target. Then, the primitive suspends in state *B* waiting for an event from the socket channel (*socket_event*) indicating that the packet can be retrieved. Finally, the retrieved data is handled by executing the operations included in the update function, moving from *B* to the final state *C*. Timing annotation is performed by exploiting the *time* parameter in the primitives and managing the time information in the handling code of the received data for implementing, for example, the *loosely timed* models.
- *Non-blocking interface.* Figure 22.11b, c show the EFSM models of the non-blocking primitives, which are composed of two states only. Primitives such as `nb_transport_fw()` and `nb_transport_bw()` perform the required operation as soon as they are called, and they immediately reach the final state in the corresponding EFSM. The caller process is informed if the non-blocking primitive succeeded by looking at its return value. Timing annotation is still performed by exploiting the *time* parameter in the primitives, while parameter *phase* is exploited for implementing more accurate communication protocols, such as the four phases *approximately timed*.

Several TLM communication protocols can be modeled by using the TLM primitives previously described, and their EFSM models can be represented by sequentially composing the EFSMs of the involved primitives. Starting from the EFSM models, the mutation model for the communication protocols is defined by following the next steps:

1. Identify a set of design errors typically introduced during the design of TLM communication protocols.
2. Identify a fault model to introduce faults (i.e., mutations) in the EFSM representations of the TLM-2.0 primitives.
3. Identify the subset of faults corresponding to the design errors identified at step 1.
4. Define mutant versions of the TLM-2.0 communication primitives implementing the faults identified at step 3.

Based on the expertise gained about typical errors made by designers during the creation of a TLM description, the following classes of design errors have been identified:

1. Deadlock in the communication phase
2. Forgetting to use communication primitives (e.g., the TLM communication primitive `nb_transport_bw()` for completing transaction phases, before initiating a new transaction is not called)

3. Misapplication of TLM operations (e.g., setting a *write* command for reading data instead of *read*)
4. Misapplication of blocking/non-blocking primitives
5. Misapplication of timed/untimed primitives
6. Erroneous handling of the generic payload (e.g., failing to set or read the packet fields)
7. Erroneous polling mechanism (e.g., infinite loop)

Other design errors could be added to the previous list to expand the proposed mutation model without altering the methodology. Each of the previous error classes has been associated with at least a mutation of the EFSM models representing TLM primitives, as described in the next paragraphs.

According to the classification of errors that may affect the specification of finite-state machine, proposed by Chow [32], different fault models have been defined for perturbing FSMs [25, 89]. They target, generally, Boolean functions labeling the transitions and/or transition's destination states. Mutated versions of an EFSM can be generated in a similar way, by modifying the behavior of enabling and update functions and/or changing the destination state of transitions.

Hereafter, we present an example of how the EFSM of Fig. 22.11a can be perturbed to generate mutant versions of the TLM primitive according to the design errors previously summarized. Figure 22.12 shows how such kinds of mutations are used to affect the behavior of primitive `b_transport()`. Numbers reported in the bottom right part of each EFSM identify the kind of design errors modeled by the mutation with respect to the previous classification.

Mutations on destination states. Changing the destination state of a transition allows us to model design errors #2, #4, and #7. For example, let us consider Fig. 22.12. Cases (a)–(d) show mutated versions of the EFSM that affect the destination state of the transition. Mutation (a) models the fact that the designer forgets to call `b_transport()` (design error #2), while (b) models the misapplication of a non-blocking primitive instead of a blocking one, since the wait on channel event is bypassed (design error #4). Cases (c) and (d) model two different incorrect uses of the polling mechanism (design error #7).

Mutations on enabling functions. Mutation on the truth value of enabling functions model design errors of type 1 and 4. For example, Fig. 22.12e shows a mutated version of the EFSM corresponding to primitive `b_transport()`, where the transition from A to B is never fired and B is never reached. Such a mutation corresponds to a deadlock in the communication protocol (design error #1), for example, due to a wrong synchronization among modules with the socket channel. The primitive can also be mutated as shown in case (f), which corresponds to using a non-blocking instead of a blocking primitive, since the wait in `B` for the channel event is prevented by an always true enabling function (design error #4).

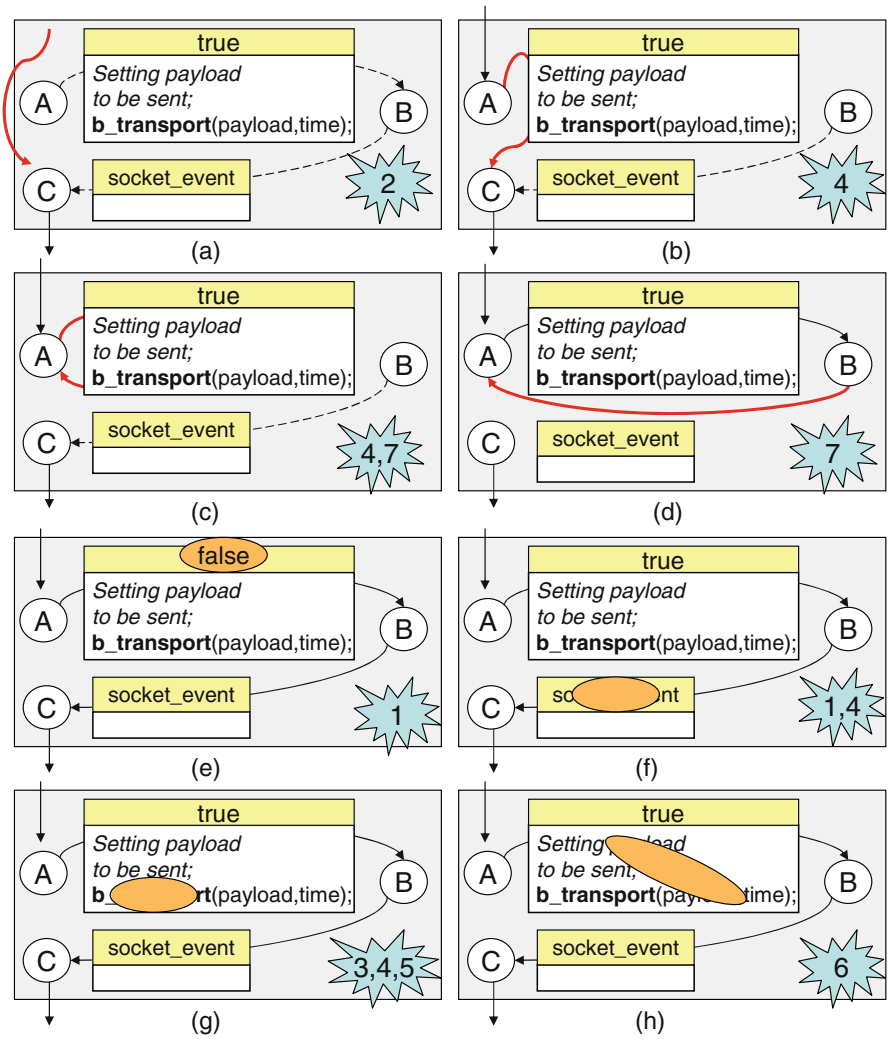


Fig. 22.12 Mutations on EFSM representing the TLM-2.0 primitive b_transport()

Mutations on update functions. Changing the operations performed in the update functions allows us to model design errors #3, #4, #5, and #6. Mutation on operations (shown in case (g)) corresponds to a misapplication of the communication primitives, like, for example, calling a transaction for writing instead of a transaction for reading (design error #3), a `b_transport()` instead of an `nb_transport()` (design error #4), setting the time parameter instead of not setting it (design error #5). On the other hand, mutations on data included in the payload packets (shown in cases (h)) model design errors corresponding to an erroneous handling of the payload packet (design error #6).

22.6.2 Property Qualification

In the last years, research topics investigated how to assess the quality and the comprehensiveness of a set of properties to increase the efficiency and effectiveness of assertion-based verification. Three approaches have emerged:

1. Detection of properties that *pass vacuously*. Properties are vacuously satisfied if they hold in a model and can be strengthened without causing them to fail. Such properties may not cover any behavior of the golden model thus they can lead to a false sense of safety. The vacuous satisfaction points out problems either in property or in environment definition or in the model implementation.
2. Analysis of the *completeness* of a set of properties. A set of properties could be incomplete since some requirements could be only partially formalized into properties. As a consequence behaviors uncovered by properties can exist, so implementation could be wrong even if it satisfies all the defined properties.
3. Identification of *over-specification*. The set of properties could be over-specified if it contains properties that can be derived as logical consequence of other properties. For example, it is possible to define a property whose coverage is a subset of the coverage of another defined property. Thus, all behaviors modeled by the first property are also modeled by the latter. The presence of such over-specification yields the verification time to be longer than it is required to be.

Current approaches to *vacuity analysis*, rely on the pioneering work of Beer et al. [10], where a formula φ is said to pass vacuously in a model M if it passes in M, and there is a sub-formula ψ of φ that can be changed arbitrarily without affecting the passing of φ. All of them, generally, exploit formal methods to search for an *interesting witness*, proving that a formula does not pass vacuously [7, 10, 11, 69, 70]. In this context, an interesting witness is a path showing one instance of the truth of the formula φ, on which every important sub-formula affects the truth of φ. Such approaches are, generally, as complex as model checking, and they require to define and model check further properties obtained from the original ones by substituting their sub-formulas in some way, thus sensibly increasing the verification time.

The analysis of the *completeness* of a set of properties addresses the question of whether enough properties have been defined. This is generally evaluated by computing *property coverage*, whose intent is to measure the percentage of DUV behaviors captured by properties. Current approaches for property coverage can be divided into two categories: mutant based [27–29, 60, 65, 71, 73, 103] and implementation based [66, 82, 102]. The first references rely on a form of mutation coverage that requires perturbing the design implementation before evaluating the property coverage. In particular, [71] gives a good theoretic background with respect to mutation of both specification and design. The latter ones estimate the property coverage by analyzing the original implementation without the need to insert perturbations. The main problem of these approaches is due to the adoption of symbolic algorithms that suffer from the state explosion problem.

Finally, regarding the problem of over-specification, the analysis can be performed by exploiting theorem proving. However, its complexity is exponential in the worst case, and it is not completely automatized, since human interaction is very often required to guide the proof. Fully automatic techniques for dealing with *over-specification removal* have not been investigated in literature, while the problem has been only partially addressed in [30]. The authors underline that given a set of properties, there can be more than one over-specified formula, and they can be mutually dependent; thus, they cannot be removed together. The authors show that finding the minimal set of properties that does not contain over-specifications is a computationally hard problem.

In general, by observing the state of the art in the literature, it appears that most of the existing strategies for property qualification rely on formal methods, which require a huge amount of spatial (memory) and temporal (time) resources, and they generally solve the qualification problem only for specific subsets of temporal logics. Given the previous drawbacks, the next section is devoted to summarizing an alternative qualification approach [39], which exploits mutation analysis and simulation to evaluate the quality of a set of formal properties with respect to vacuity [42], completeness [48], and over-specification [21].

22.6.2.1 Mutant-Based Property Qualification

As shown on top of Fig. 22.13, the basic ingredients for the mutant-based property qualification methodology are the model of the DUV, a corresponding set of formal properties that hold on the model and, if necessary, the description of the environment where the DUV is embedded in. The approach is independent from the abstraction level of the DUV and the logics used for the definition of the properties. Properties are then converted into checkers, i.e., monitors connected to the DUV that allow checking the satisfiability of the corresponding properties by simulation. Checkers can be easily generated by adopting automatic tools, like, for example, IBM FoCs [2] or radCHECK [38].

According to the central part of Fig. 22.13, mutants are injected into either the DUV or the checkers to generate their corresponding faulty versions. Faulty checker implementations are generated for addressing vacuity analysis, while faulty DUV implementations are necessary for measuring property coverage and detecting cases of over-specification.

Entering in the details, the **vacuity analysis** for a property φ is carried on in relation to the effect of mutants that affect the sub-formulas of φ. In particular, the methodology works as follows:

1. Given a set of properties that are satisfied by the DUV, a set of *interesting mutants* is injected in the corresponding checkers. Intuitively, an interesting mutant perturbs the checker's behavior similar to what happens when a sub-formula ψ is substituted by **true** or **false** in φ according to the vacuity analysis approach proposed in [11]. Thus, for each minimal sub-formula (i.e., each atomic proposition) ψ of φ, a mutant is injected in the corresponding checker, such that the signal storing the value of ψ is stuck at true or stuck at false, respectively,

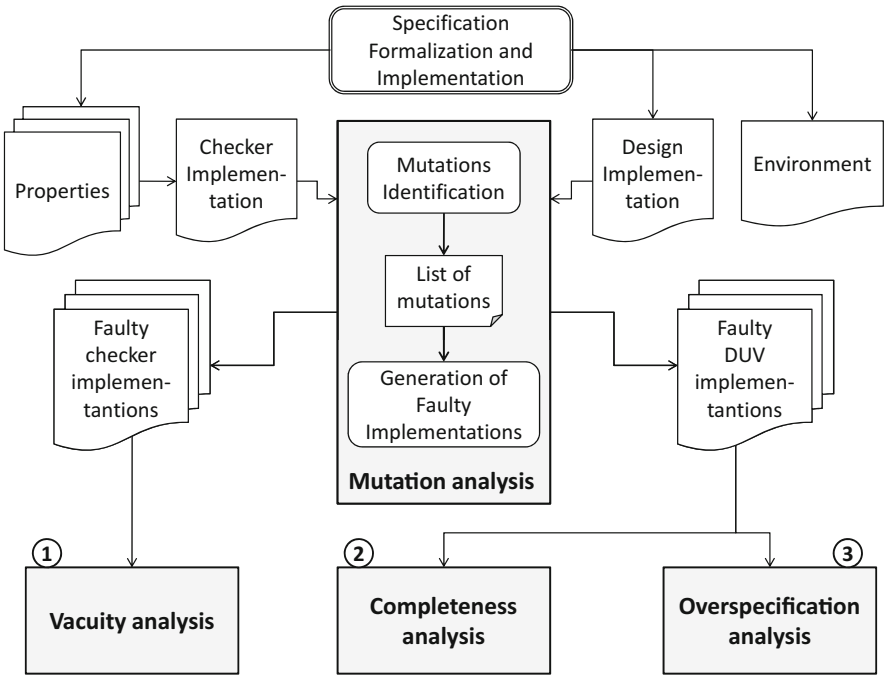


Fig. 22.13 Overview of the property qualification methodology

when ψ has a negative or a positive polarity. (Intuitively, in a logic with polarity, a formula has positive polarity if it is preceded by an even number of not operators; otherwise, it has negative polarity).

2. The faulty checkers are connected to the DUV and the related environment. Then, testbenches are used to simulate the DUV. The vacuity analysis relies on the observation of the simulation result. A checker failure due to the effect of an interesting mutant f corresponds to proving that the sub-formula ψ perturbed by f affects the truth value of φ. Consequently, the sequence of values generated by the testbench that causes the checker failure is an interesting witness proving that φ is not vacuous with respect to its sub-formula ψ. On the contrary, a mutant that does not cause checker failures (i.e., an undetected mutant) must be analyzed to determine if either the property is vacuous with respect to the corresponding sub-formula or the vacuity alert is due to the inefficiency of the testbenches that cannot detect the mutant during the simulation. The latter happens when there exists a test sequence (i.e., an interesting witness) for detecting the mutant, but the testbenches are not able to generate it. In case of an undetected mutant, the verification engineer can manually investigate the cause of undetectability to discriminate between a vacuous property and a low-quality testbench. However, it is also possible to make the disambiguation in an automatic way, by means of a formal approach. In fact, a new property φ' can be generated from φ, where

the sub-formula ψ inside φ is substituted with either *true* or *false*, depending on the polarity of ψ, to reproduce the same effect caused by injecting the mutant f on the checker of φ. Then, the satisfiability of φ' is verified on the DUV by using a model checker. If the model checker returns a successful answer, it implies that φ is vacuous, because ψ does not affect φ; otherwise it means that φ is not vacuous while the testbench is ineffective. In this second case, the counterexample generated by the model checker represents the input sequence that must be added to the testbench to prove the non-vacuity of φ by simulation.
3. Finally, the analysis of interesting mutants that actually correspond to vacuous passes allows the verification engineer to determine the exact cause of the vacuity, which can be either an error in the property, a too strict environment for the DUV, or an error in the model of the DUV itself that does not implement correctly the intended specification.

At the end of the vacuity analysis, it is possible to measure the **degree of completeness** of the remaining properties on the basis of their property coverage. According to the theoretical basis described in [48], property coverage is computed by measuring the capability of a set of properties to detect mutants that perturb the DUV. A low property coverage is then a symptom of a low degree of completeness. In particular, if a mutant that perturbs the functionality of the DUV does not affect at least one property in the considered set, it means that this set of properties is unable to distinguish between the faulty and the fault-free implementations of the DUV, and thus it is incomplete. In this case, a new property covering the fault should be added. On the contrary, the fact that at least one property fails in the presence of each mutant affecting the outputs of the DUV implementation represents a positive feedback about the quality of the property set. Nevertheless, the quality of the mutant model is the key aspect of the overall methodology. A low-quality set of mutants negatively impacts the overall methodology, such that achieving 100% property coverage provides a false sense of security when mutant injection is inadequate.

Independently from the adopted mutant model, the computation of the property coverage consists of two phases:

1. *Generation of faulty DUV implementations.* Perturbations of the design implementation are generated by automatically injecting mutants inside the DUV model. The obtained mutant list must include only *detectable mutants*, which are mutants that, for at least one input sequence, cause at least one output of the faulty implementation to differ from the corresponding output of the fault-free implementation. Only detectable mutants are considered to achieve an accurate estimation of the golden model completeness because undetectable mutants cannot cause failures on the properties since they do not perturb the outputs of the DUV. The set of detectable mutants can be identified by simulating the DUV with either manual testbenches or by using an automatic test pattern generator.
2. *Property coverage analysis.* The presence of a detectable mutant implies that the behavior of the faulty implementation differs from the behavior of the fault-free

implementation. Thus, while the defined properties are satisfied by the fault-free implementation, at least one of them should be falsified if checked on a faulty implementation. The property coverage is then measured by the following formula:

$$PC = \frac{\text{number of mutants detected by the properties}}{\text{number of detectable mutants}}$$

The computation of PC can be done by using a formal approach, i.e., by model checking the properties in the presence of each mutant, or by means of a dynamic strategy, i.e., by simulating the faulty DUV connected to the checkers corresponding to the properties under analysis. The formal approach is generally unmanageable for a large number of mutants, but the higher scalability provided by the dynamic simulation is paid in terms of exhaustiveness. In fact, as for the case of vacuity analysis, an undetected mutant during simulation can be due to the ineffectiveness of the testbenches rather than an incompleteness of the properties. Thus, formal analysis is restricted to the few mutants that remain undetected after simulation.

3. In case of a low property coverage, the verification engineer is guided in the definition of new properties by analyzing the area of the DUV including mutants that do not affect any property till the desired degree of completeness is achieved.

When the achieved degree of completeness is satisfactory (this measure depends on the design team standards), a last process, still based on the property coverage, can be applied for capturing the case of **over-specification**, i.e., the presence of properties that can be removed from the final set because they are covering the same behaviors covered by other properties included in the same set [21].

22.7 Automatic Stimuli Generation

The main purpose of dynamic verification is increasing the confidence of designers in the ES behavior by creating stimuli and evaluating them in terms of adequacy criteria, e.g., coverage metrics. In this context, an effective stimuli generation is at the basis of a valuable functional qualification. Actual value inputs may be either automatically generated or developed by engineers as stimuli suites. Stimuli generation techniques fall into three main categories: *concrete execution*, *symbolic execution*, and *concolic execution*. Concrete execution is based on random, probabilistic, or genetic techniques [79]. It is not an exhaustive approach, but it allows to reach deep states of the system space by executing a large number of long paths. Symbolic execution [68] represents an alternative approach for overcoming concrete execution limitations, where an executable specification is simulated using symbolic variables, and a decision procedure is used to obtain concrete values for inputs. Such approaches suffer from solver limitations in handling either the complexity of formulas or data structures or the surrounding

execution environment. Such limitations have been recently addressed by proposing *concolic execution* [77, 93] that mingles concrete and symbolic executions and, whenever necessary, simplifies symbolic constraints with corresponding concrete values. However, module execution is still represented as a symbolic execution tree, growing exponentially in the number of the maintained paths, states, and conditions, thus incurring in state space explosion.

Several tools on the market adopt these approaches and provide the user with automatic stimuli generation addressing coverage metrics. DART [54] combines random stimuli generation with symbolic reasoning to keep track of constraints for executed control flow paths. CUTE [93] is a variation of the DART approach addressing complex data structures and pointer arithmetic. EXE [23] and KLEE [22] are frameworks for symbolic execution, where the second symbolically executes LLVM [72] bytecode. PEX [99] is an automated structural testing generation tool for .NET code developed at Microsoft Research. [77] describes a hybrid concolic stimuli generation approach for C programs, interleaving random stimuli generation with bounded exhaustive symbolic exploration to achieve better coverage. However, it cannot selectively and concolically execute paths in a neighborhood of the corner cases.

22.7.1 EFSM-Based Stimuli Generation

This section presents an EFSM-based concolic stimuli generation approach for ES. The approach is based on an EFSM model of the ES and leads to traverse a *target transition t* (i.e., not-yet-traversed transition) by integrating concrete execution and a symbolic technique that ensures exhaustiveness along specific paths leading to the target transition t.

Algorithm 1 is a high-level description of the proposed concolic approach. It takes as inputs the EFSM model and two timeout thresholds: (i) overall timeout, i.e., the maximum execution time of the algorithm (*MaxTime*), and (ii) inactivity timeout, i.e., the maximum execution time the long-range concrete technique can spend without improving transition coverage (*InaTime*).

Algorithm 1 The EFSM-based concolic algorithm for stimuli generation

```
1  procedure EFSMStimuliGen(EFSM, MaxTime, InaTime)
   input: embedded-application model EFSM, overall timeout MaxTime, inactivity timeout
          InaTime
   output: set of stimuli Stimuli
2  Stimuli ← ∅; RInf ← ∅;
3  DInf ← DependencyAnalysis(EFSM);
4  while elapsed time < MaxTime do
5  |   while inactivity timeout InaTime not expired do
6  |   |   (stimulus, reach) ← LongRangeSearch(EFSM, RInf);
7  |   |   Stimuli ← Stimuli ∪ {stimulus}; RInf ← RInf ∪ {reach};
8  |   (stimulus, reach) ← GuidedWideWidthSearch(EFSM, RInf, DInf);
9  |   Stimuli ← Stimuli ∪ {stimulus}; RInf ← RInf ∪ {reach};
10 return Stimuli
```

RInf keeps track of the EFSM configurations used for (re-)storing system status whenever the algorithm switches between the symbolic and concrete techniques. At the beginning, both *Stimuli* and *RInf* are empty (line 2). The algorithm identifies the dependencies between internal and input variables and EFSM paths. *EFSM transitions* allow to determine dependency information (*DInf*, line 3), used in the following corner-case-oriented symbolic phases, when further dynamic analysis between *EFSM paths* is performed. Such a dependency analysis selectively chooses a path for symbolic execution whenever the concrete technique fails in improving transition coverage of the EFSM. The stimuli generation runs until the specified overall timeout expires (line 4–5). First, the algorithm executes a long-range concrete technique (line 6), then a symbolic wide-width technique, which exploits the Multi-Level Back Jumping (MLBJ) (see Sect. 22.7.1.3) to cover corner cases (line 7). The latter starts when the transition coverage remains steady for the user-specified inactivity timeout (line 5). The algorithm reverts back to the long-range search as soon as the wide-width search traverses a target transition. The output is the generated stimuli set (line 9). The adopted long-range search (line 6) exploits constraint-based heuristics [40] that focus on the traversal of just one transition at a time. Such approaches scale well with design size and, significantly, improve the bare pure random approach. The following sections deepen the single steps of the algorithm.

22.7.1.1 Dependency Analysis

Without a proper dependency analysis, the stimuli generation engine wastes considerable effort in the exploration of uninteresting parts of the design. Thus, the proposed approach focuses on dependencies of enabling functions, i.e., control part, on internal variables. As a further motivating example, consider the EFSM in Fig. 22.2. Let t_8 be the target transition. We want to compare paths $\pi_1 = t_2 :: t_6$ and $\pi_2 = t_3$, where $t :: t'$ denotes the concatenation of transitions t and t'. The enabling function of t_8 involves the variables "ova" and "pva." Both are defined along π_1 by means of primary inputs. Along π_2 only "pva" is defined by means of primary inputs. Thus, to traverse t_8, MLBJ will select π_1 instead of π_2, since t_8 enabling function is more likely to be satisfied by the symbolic execution of π_1 rather than of π_2. We will consider again this example at the end of the section.

We approximate dependencies as *weights*. Indirect dependencies, as the dependency of d_2 on i_1 in the sequence of assignments "$d_1 := i_1 + i_2;\ d_2 := d_1 + i_3$," are approximated as *flows* of weights between assignments. Given a target transition $\bar{t}$, we map each path ending in $\bar{t}$ to a nonnegative weight representing data control dependencies. Intuitively, the higher the weight, the greater is the dependency of $\bar{t}$'s enabling function, i.e., $e_{\bar{t}}$, on inputs read along such a path, and the higher is the likelihood that its symbolic execution leads to the satisfaction of $e_{\bar{t}}$. An initial weight is assigned to $e_{\bar{t}}$, then we let it "percolate" backward along paths. Each transition lets a fraction of the received weight percolate to the preceding nodes and retains the remaining fraction. The weight associated with a path π is defined as *the sum of weights retained by each transition of* π. The ratio of the weight retained by a transition is defined by its update function.

22.7.1.2 Snapshots of the Concrete Execution

The ability of saving the EFSM configurations allows the system to be restored during the switches between concrete and symbolic phases. This avoids the time consuming re-execution of stimuli. Algorithm 1 keeps trace of the reachability information, i.e., *RInf*, and maintains a cache of snapshots of the concrete execution. Each time a stimulus is added to the set of stimuli, the resulting configuration is stored in memory and explicitly linked to the reached state. The wide-width technique searches feasible paths that both start from an intermediate state of the execution and lead to the target transition. Moreover, during the MLBJ, for a given configuration and target transition, many paths are checked for feasibility, as described in Sect. 22.7.1.3. Thus, caching avoids the cost of recomputing configurations for each checked path. Both the time and memory requirements of each snapshot are proportional to the size of D (see definition in Sect. 22.4.1.1).

22.7.1.3 Multilevel Back Jumping

When the long-range concrete technique reaches the inactivity time-out threshold, the concolic algorithm switches to the weight-oriented symbolic approach; see line 7 in Algorithm 1. Typically, some hard-to-traverse transitions, whose enabling functions involve internal variables, prevent the concrete technique goes further in the exploration. In this case, the MLBJ technique is able to selectively address

Algorithm 2 The core of the MLBJ technique

```
1  procedure MLBJ(t̄, timeout)
   input: target transition t̄ ∈ T, timeout
   output: stimuli for t̄, in case empty
2  Let w^0 be the initial weight tuple such that
3  ∀d_i ∈ D . w_i^0 = { 1  if d_i occurs in EnablingFunction(t̄),
                      { 0  otherwise.
4  p ← {(t̄, w^0, 0, 0)};
5  while elapsed time < timeout do
6      (t :: π, w, r', r) ← remove_top(p)
7      // w = P*(t :: π, w^0), r' = R*(t :: π, w^0),
8      // r = R*(π, w^0)
9      if t :: π is satisfiable then
10         if r < r' then
11             foreach configuration ⟨src(t), k⟩ do
12                 if k ∧ t :: π is satisfiable then
13                     return stimuli for t̄;
14         foreach {t' ∈ T | dst(t') = src(t)} do
15             w' = P*(t' :: t :: π, w^0);
16             r'' = R*(t' :: t :: π, w^0);
17             push(p, (t' :: t :: π, w', r'', r'));
```

paths, with high dependency on inputs, i.e., high retained weight, for symbolically executing them. Such paths are leading from an intermediate state of the execution to the target transition; thus the approach is exhaustive in a neighborhood of the corner case.

Algorithm 2 presents a description of the core of the MLBJ procedure. A transition $\bar{t}$ is selected in the set of target transition, and then a progressively increasing neighborhood of $\bar{t}$ is searched for paths π leading to $\bar{t}$ and having maximal retained weight, i.e., $R^*(\pi, w^0)$. If the approach fails, another target transition is selected in the set, and the procedure is repeated.

Describing in an elaborate way, a visit is started from $\bar{t}$ that proceeds backward in the EFSM graph. The visit uses a priority queue p, whose elements are paths that end in $\bar{t}$. In particular, each path of p is accompanied by its weight tuple, i.e., $w = P^*(\pi, w^0)$ and retained weights, i.e., $r = R^*(\pi, w^0)$ and $r' = R^*(t :: \pi, w^0)$. At the beginning, the queue p contains only $\bar{t}$ and the associated initial weight w^0 (lines 2–3); no weight is initially retained (line 4). At each iteration, a path $t :: \pi$ with maximal retained weight is removed from p (line 6). The decision procedure is used to check if the path $t :: \pi$ can be proved unsatisfiable in advance (line 9), e.g., it contains clause conflicts. In this case $t :: \pi$ is discarded so the sub-tree preceding $t :: \pi$ will not be explored. Otherwise, if the transition t has yielded a positive retained weight (line 10), and then for each configuration associated with the source state of t, the decision procedure checks the existence of a sequence of stimuli that leads to the traversal of $t :: \pi$ and thus of $\bar{t}$ (lines 12–13). In particular, the path constraint is obtained by the identified EFSM path, i.e., $t :: \pi$, and the concrete values of the internal variables, i.e., k, (line 12). In case a valid sequence of stimuli has not been identified, for each transition t' that precedes t, the path $t' :: t :: \pi$ is added to the priority queue p (lines 14–17).

22.8 Conclusion

This chapter focused on the realization of an effective semiformal ABV environment in a Model-Driven Design Framework. First it provided a general introduction to Model-Driven Design and Assertion-Based Verification concepts and related formalisms and then a more detailed view on the main challenges concerning their combined use. Assertion-based verification can hypothetically provide an exhaustive answer to the problem of design correctness, but from the practical point of view, this is possible only if (1) the design under verification is stimulated with testbenches that generate the set of all possible input stimuli and (2) a complete set of formal properties is defined that totally captures the designer's intents. Therefore the chapter addressed assertion definition and automatic generation of checkers and stimuli.

The key ingredient for an effective design and verification framework based on MDD and ABV, is represented by the possibility of defining a model of the desired system and then automatically deriving the corresponding simulatable description to be used for virtual prototyping. This aspect was addressed in the chapter by using

automata-based formalisms, together with bottom-up and top-down flows, whose combined adoption allows the generation of a homogeneous simulatable description of the overall system.

Finally, the problem of property qualification was addressed by discussing about property vacuity, completeness and over-specification.

References

1. 3S Software (2012) CoDeSys. http://www.3s-software.com
2. Abarbanel Y, Beer I, Gluhovsky L, Keidar S, Wolfsthal Y (2000) FoCs: automatic generation of simulation checkers from formal specifications. In: Proceedings of international conference on computer aided verification (CAV), pp 538–542
3. Aerospace Valley (2012) TOPCASED project. http://www.topcased.org
4. Alagar V, Periyasamy K (2011) Extended finite state machine. In: Specification of software systems, texts in computer science. Springer, London, pp 105–128. DOI 10.1007/978-0-85729-277-3_7
5. Alexander RT, Bieman JM, Ghosh S, Bixia J (2002) Mutation of Java objects. In: Proceedings of IEEE ISSRE, pp 341–351
6. Alur R, Dill DL (1994) A theory of timed automata. Theoret Comput Sci 126(2):183–235
7. Armoni R, Fix L, Flaisher A, Grumberg O, Piterman N, Tiemeyer A, Vardi M (2003) Enhanced vacuity detection in linear temporal logic (CAV). In: International conference on computer aided verification, vol 2725. Springer, Berlin/Heidelberg, pp 368–380
8. Atego (2012) ARTiSAN. http://www.atego.com/products/artisan-studio
9. Batth SS, Vieira ER, Cavalli A, Umit Uyar M (2007) Specification of timed EFSM fault models in SDL. In: Proceedings of FORTE, pp 50–65
10. Beer I, Ben-David S, Eisner U, Rodeh Y (1997) Efficient detection of vacuity in ACTL formulas. In: International conference on computer aided verification (CAV), vol 1254, pp 279–290
11. Beer I, Ben-David S, Eisner C, Rodeh Y (2001) Efficient detection of vacuity in temporal model checking. Form Methods Syst Des 18(2):141–163
12. Belli F, Budnik CJ, Wong WE (2006) Basic operations for generating behavioral mutants. In: Proceedings of IEEE ISSRE, pp 10–18
13. Black P, Okun V, Yesha Y (2000) Mutation operators for specifications. In: Proceedings of IEEE ASE, pp 81–88
14. Bombieri N, Fummi F, Guarnieri V, Pravadelli G (2014) Testbench qualification of systemc TLM protocols through mutation analysis. IEEE Trans Comput 63(5):1248–1261
15. Bombieri N, Fummi F, Pravadelli G, Hampton M, Letombe F (2009) Functional qualification of TLM verification. In: Design, automation test in Europe conference exhibition, DATE'09, pp 190–195. DOI 10.1109/DATE.2009.5090656
16. Borrione D, Liu M, Morin-Allory K, Ostier P, Fesquet L (2005) On-line assertion-based verification with proven correct monitors. In: Proceedings of international conference on information and communications technology (ICICT), pp 125–143
17. Boulé M, Zilic Z (2008) Automata-based assertion-checker synthesis of PSL properties. ACM Trans Des Autom Electron Syst 13:1–21. http://doi.acm.org/10.1145/1297666.1297670
18. Boutekkouk F, Benmohammed M, Bilavarn S, Auguin M et al (2009) UML 2.0 profiles for embedded systems and systems on a chip (SoCs). J Object Technol 8(1):135–157. DOI 10.5381/jot.2009.8.1.a1
19. Bradbury JS, Cordy JR, Dingel J (2006) ExMan: a generic and customizable framework for experimental mutation analysis. In: Proceedings of IEEE ISSRE, pp 4–9
20. Bradbury JS, Cordy JR, Dingel J (2006) Mutation operators for concurrent Java (J2SE 5.0). In: Proceedings of IEEE ISSRE, pp 11–11

21. Brait S, Fummi F, Pravadelli G (2005) On the use of a high-level fault model to analyze logical consequence of properties. In: Proceedings of ACM/IEEE international conference on formal methods and models for co-design, MEMOCODE, pp 221–230
22. Cadar C, Dunbar D, Engler D (2008) KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of USENIX symposium on operating systems design and implementation (OSDI)
23. Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR (2006) EXE: a system for automatically generating inputs of death using symbolic execution. In: ACM conference on computer and communications security, pp 322–335
24. Cadence (2012) Assertion-based verification. http://www.cadence.com/products/fv/pages/abv_flow.aspx
25. Cheng KT, Jou JY (1990) A single-state-transition fault model for sequential machines. In: IEEE ICCAD'90, pp 226–229
26. Cheung P, Forin A (2007) A C-language binding for PSL. In: Proceedings of international conference on embedded software and systems (ICESS). Springer, pp 584–591
27. Chockler H, Kupferman O, Kurshan R, Vardi M (2001) A practical approach to coverage in model checking. In: Proceedings computer aided and verification, pp 66–78
28. Chockler H, Kupferman O, Vardi M (2006) Coverage metrics for formal verification. Int J Softw Tools Technol Transfer (STTT) 8:373–386
29. Chockler H, Kupferman O, Vardi M (2006) Coverage metrics for temporal logic model checking. Formal Methods Syst Des 28:189–212
30. Chockler H, Strichman O (2007) Easier and more informative vacuity checks. In: Proceedings ACM/IEEE international conference on formal methods and models for codesign, pp 189–198
31. Choi BJ, DeMillo RA, Krauser EW, Martin RJ, Mathur AP, Pan AJOH, Spafford EH (1989) The Mothra tool set (software testing). In: Proceedings of IEEE HICSS, vol 2, pp 275–284
32. Chow T (1978) Testing software design modeled by finite state machines. IEEE Trans Softw Eng 4(3):178–187
33. Dahan A, Geist D, Gluhovsky L, Pidan D, Shapir G, Wolfsthal Y, Benalycherif L, Kamidem R, Lahbib Y (2005) Combining system level modeling with assertion-based verification. In: Proceedings of international symposium on quality of electronic design (ISQED), pp 310–315
34. Das S, Mohanty R, Dasgupta P, Chakrabarti P (2006) Synthesis of system verilog assertions. In: Proceedings of design, automation & test in Europe conference & exhibition (DATE), vol 2, pp 1–6
35. De Simone R, André C (2006) Towards a "synchronous reactive" UML profile? Int J Softw Tools Technol Transfer 8(2):146–155
36. Delamaro ME, Maldonado JC (1996) Proteum – a tool for the assessment of test adequacy for C programs. In: PCS'96, pp 79–95
37. DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: help for the practicing programmer. IEEE Comput 11(4):34–41
38. Di Guglielmo G, Di Guglielmo L, Foltinek A, Fujita M, Fummi F, Marconcini C, Pravadelli G (2013) On the integration of model-driven design and dynamic assertion-based verification for embedded software. J Syst Softw 86(8):2013–2033. DOI 10.1016/j.jss.2012.08.061
39. Di Guglielmo L, Fummi F, Pravadelli G (2009) The role of mutation analysis for property qualification. In: IEEE/ACM international conference on formal methods and models for co-design, MEMOCODE, pp 28–35
40. Di Guglielmo G, Fummi F, Pravadelli G, Soffia S, Roveri M (2010) Semi-formal functional verification by EFSM traversing via NuSMV. In: Proceedings of IEEE international high level design validation and test workshop (HLDVT), pp 58–65
41. Di Guglielmo L, Fummi F, Orlandi N, Pravadelli G (2010) DDPSL: an easy way of defining properties. In: 2010 IEEE international conference on computer design (ICCD), pp 468–473
42. Di Guglielmo L, Fummi F, Pravadelli G (2010) Vacuity analysis for property qualification by mutation of checkers. In: Design, automation test in Europe conference exhibition (DATE), pp 478–483

43. Di Guglielmo L, Fummi F, Pravadelli G, Stefanni F, Vinco S (2012) A formal support for homogeneous simulation of heterogeneous embedded systems. In: IEEE international symposium on industrial embedded systems (SIES), pp 211–219
44. Di Guglielmo L, Fummi F, Pravadelli G, Stefanni F, Vinco S (2013) UNIVERCM: the universal versatile computational model for heterogeneous system integration. IEEE Trans Comput 62(2):225–241
45. Ebeid E, Fummi F, Quaglia D (2015) HDL code generation from UML/MARTE sequence diagrams for verification and synthesis. Des Autom Embed Syst 19(3):277–299. DOI 10.1007/s10617-014-9158-1
46. Ebeid E, Fummi F, Quaglia D (2015) Model-driven design of network aspects of distributed embedded systems. IEEE Trans Comput Aided Des Integr Circuits Syst 34(4):603–614
47. Ebert C, Jones C (2009) Embedded software: facts, figures, and future. Computer 42(4): 42–52
48. Fedeli A, Fummi F, Pravadelli G (2007) Properties incompleteness evaluation by functional verification. IEEE Trans Comput 56(4):528–544
49. Ferrari A, Gaviani G, Gentile G, Stara G, Romagnoli G, Thomsen T (2004) From conception to implementation: a model based design approach. In: Proceedings of IFAC symposium on advances in automotive control
50. Ferro L, Pierre L (2010) ISIS: runtime verification of TLM platforms. Adv Des Methods Model Lang Embed Syst SoCs 63:213–226
51. Foster H, Krolnik A, Lacey D (2004) Assertion-based design. Springer, New York
52. Foster H, Larsen K, Turpin M (2006) Introducing the new accellera open verification library standard. In: Proceedings of design and verification conference (DVCON)
53. Gentleware (2012) Poseidon for UML embedded edition. http://www.gentleware.com/uml-software-embedded-edition.html
54. Godefroid P, Klarlund N, Sen K (2005) DART: directed automated random testing. In: Proceedings of ACM SIGPLAN conference on programming language, design, and implementation (PLDI), pp 213–223
55. Graaf B, Lormans M, Toetenel H (2003) Embedded software engineering: the state of the practice. IEEE Softw 20(6):61–69
56. Di Guglielmo L, Fummi F, Pravadelli G, Stefanni F, Vinco S (2011) UNIVERCM: The UNIversal VERsatile computational model for heterogeneous embedded system design. In: Proceedings of IEEE HLDVT, pp 33–40
57. HAL – Inria (2012) Gaspard2 UML profile documentation. http://hal.inria.fr/inria-00171137/en
58. Henzinger T (1996) The theory of hybrid automata. In: Logic in computer science (LICS). IEEE Computer Society, New Brunswick, pp 278–292
59. Hiller M (2000) Executable assertions for detecting data errors in embedded control systems. In: Proceedings of IEEE international conference on dependable systems and networks (DSN), pp 24–33
60. Hoskote Y, Kam T, Ho P, Zhao X (1999) Coverage estimation for symbolic model checking. In: Proceedings ACM/IEEE design automation conference, pp 300–305
61. Hyunsook D, Rothermel G (2006) On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans Softw Eng 32(9): 733–752
62. IAR Systems (2012) IAR visualSTATE. http://www.iar.com/Products/IAR-visualSTATE/
63. IBM (2012) Rational Rhapsody. http://www.ibm.com/software/awdtools/rhapsody
64. IEEE Computer Society (2010) IEEE Standard for Property Specification Language (PSL) (IEEE Std 1850-2010)
65. Jayakumar N, Purandare M, Somenzi F (2003) Dos and don'ts of CTL state coverage estimation. In: Proceedings of design automation conference (DAC)
66. Katz S, Grumberg O (1999) Have I written enough properties? – a method of comparison between specification and implementation. In: Proceedings ACM advanced research working conference on correct hardware design and verification methods. Springer, pp 280–297

67. Kim M, Kim Y, Kim H (2011) A comparative study of software model checkers as unit testing tools: an industrial case study. IEEE Trans Softw Eng 37(2):146–160
68. King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394
69. Kupferman O, Vardi MY (1999) Vacuity detection in temporal model checking. In: Conference on correct hardware design and verification methods, pp 82–96
70. Kupferman O, Vardi M (2003) Vacuity detection in temporal model checking. Int J Softw Tools Technol Transfer 4(2):224–233
71. Kupferman O, Li W, Seshia S (2008) A theory of mutations with applications to vacuity, coverage, and fault tolerance. In: Proceedings IEEE international conference on formal methods in computer-aided design
72. Lattner C, Adve V (2005) The LLVM compiler framework and infrastructure tutorial. In: Proceedings of international workshop on languages and compilers for high performance computing (LCPC). Springer, pp 15–16
73. Lee T, Hsiung P (2004) Mutation coverage estimation for model checking. In: Proceedings international symposium on automated technology for verification and analysis, pp 354–368
74. Lettnin D, Nalla P, Ruf J, Kropf T, Rosenstiel W, Kirsten T, Schonknecht V, Reitemeyer S (2008) Verification of temporal properties in automotive embedded software. In: Proceedings of design, automation & test in Europe conference & exhibition (DATE). ACM, pp 164–169
75. Lyu MR, Zubin H, Sze SKS, Xia C (2003) An empirical study on testing and fault tolerance for software reliability engineering. In: Proceedings of IEEE ISSRE, pp 119–130
76. Ma YS, Offutt J, Kwon YR (2005) Mujava: an automated class mutation system. Softw Test Verif Reliab 15(2):97–133
77. Majumdar R, Sen K (2007) Hybrid concolic testing. In: Proceedings of IEEE international conference on software engineering (ICSE), pp 416–426
78. Mathur AP (1991) Performance, effectiveness, and reliability issues in software testing. In: COMPSAC'91, pp 604–605
79. McMinn P (2004) Search-based software test data generation: a survey. Softw Test Verif Reliab 14(2):105–156
80. Mentor Graphics (2012) Assertion-based verification . http://www.mentor.com/products/fv/methodologies/abv
81. Mischkalla F, He D, Mueller W (2010) A UML profile for SysML-based comodeling for embedded systems simulation and synthesis. In: Proceedings of workshop on model based engineering for embedded system design (MBED)
82. Mishra P, Dutt N (2002) Automatic functional test program generation for pipelined processors using model checking. In: Proceedings IEEE high-level design validation and test, pp 99–103
83. Object Management Group, Inc. (2012) MARTE resource page. http://www.omgmarte.org/
84. Object Management Group, Inc. (2012) OMG specifications. http://www.omg.org
85. Object Management Group, Inc. (2012) UML resource page. http://www.uml.org
86. Offutt AJ, Untch RH (2001) Mutation 2000: uniting the orthogonal. In: Wong WE (ed) Mutation testing for the new century. Kluwer Academic Publishers, Boston, pp 34–44
87. Offutt AJ, Rothermel G, Zapf C (1993) An experimental evaluation of selective mutation. In: ICSE'93, pp 100–107
88. Olsson T, Runeson P (2001) System level mutation analysis applied to a state-based language. In: Proceedings of IEEE ECBS, pp 222–228
89. Pinto Ferraz Fabbri SC, Delamaro ME, Maldonado JC, Masiero PC (1994) Mutation analysis testing for finite state machines. In: IEEE ISSRE'94, pp 220–229
90. Riccobene E, Scandurra P, Bocchio S, Rosti A, Lavazza L, Mantellini L (2009) SystemC/C-based model-driven design for embedded systems. ACM Trans Embed Comput Syst 8(4):1–37
91. Seger C (2006) Integrating design and verification – from simple idea to practical system. In: Proceedings of ACM/IEEE MEMOCODE, pp 161–162
92. Selic B (2003) The pragmatics of model-driven development. IEEE Softw 20(5):19–25

93. Sen K, Agha G (2006) CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Proceedings of international conference on computer aided verification (CAV). Springer, Berlin/New York, pp 419–423
94. Society IC (2009) IEEE standard for system verilog-unified hardware design, specification, and verification language (IEEE Std 1800-2009)
95. Sparx Systems (2012) Enterprise architet. http://www.sparxsystems.com.au
96. STM Products (2012) radCHECK. http://www.verificationsuite.com
97. SysML Partners (2012) SysML resource page. http://www.sysml.org
98. The MathWorks, Inc. (2012) Simulink. http://www.mathworks.com/products/simulink/
99. Tillmann N, De Halleux J (2008) Pex: white box test generation for. NET. In: Proceedings of ACM international conference on tests and proofs (TAP), pp 134–153
100. Winterholer M (2006) Transaction-based hardware software co-verification. In: Proceedings of forum on specification & design languages (FDL)
101. Xie F, Liu H (2007) Unified property specification for hardware/software co-verification. In: Proceedings of international computer software and applications conference (COMSAC), pp 483–490
102. Xu X, Kimura S, Horikawa K, Tsuchiya T (2005) Transition traversal coverage estimation for symbolic model checking. In: Proceedings ACM/IEEE international conference on formal methods and models for co-design, pp 259–260
103. Xu X, Kimura S, Horikawa K, Tsuchiya T (2006) Transition-based coverage estimation for symbolic model checking. In: Proceedings ACM/IEEE Asia and South Pacific conference on design automation, pp 1–6

CPA: Compositional Performance Analysis 23

Robin Hofmann, Leonie Ahrendts, and Rolf Ernst

Abstract

In this chapter we review the foundations Compositional Performance Analysis (CPA) and explain many extensions which support its application in design practice. CPA is widely used in automotive system design where it successfully complements or even replaces simulation-based approaches.

Acronyms

ACK	Acknowledgement
ARQ	Automatic Repeat Request
BCET	Best-Case Execution Time
BCRT	Best-Case Response Time
CAN	Controller Area Network
COTS	Commercial/Components Off-The-Shelf
CPA	Compositional Performance Analysis
DAG	Directed Acyclic Graph
DMA	Direct Memory Access
ECU	Electronic Control Unit
FIFO	First-In First-Out
MCR	Mode Change Request
SPNP	Static-Priority Non-Preemptive
SPP	Static Priority Preemptive
TWCA	Typical Worst-Case Analysis
TWCRT	Typical Worst-Case Response Time
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

R. Hofmann (✉) • L. Ahrendts • R. Ernst
Institute of Computer and Network Engineering, Technical University Braunschweig, Braunschweig, Germany
e-mail: rhofmann@ida.ing.tu-bs.de; ahrendts@ida.ing.tu-bs.de; ernst@ida.ing.tu-bs.de

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_24

Contents

23.1 Motivation

Despite the risk of overlooking critical corner cases, design verification is for the most part based on execution and test using simulation, prototyping, and the final system. Formal analysis and verification are typically used in cases where errors are particularly expensive or may have catastrophic consequences, such as in safety critical or high availability systems. Such formal methods have considerably improved in performance and usability and can be used on a broader scale to improve design quality, but they must cope with growing hardware and software architecture complexity.

The situation is similar when we consider system timing verification. Formal timing analysis methods have been around for decades, starting with early work by Liu and Layland in the 70s [24] which provided schedulability analysis and worst-case response time data for a limited set of task system classes and scheduling strategies for single processors. In the meantime, there were dramatic improvements in the scope of considered tasks systems, architectures, and timing models. One of the key analysis inputs is the maximum execution time of a task, the Worst-Case Execution Time (WCET), where there has been similar progress [51]. As in the case of function verification, progress in hardware and software architectures made analysis more challenging. In particular the dominant trend focusing Commercial/Components Off-The-Shelf (COTS) on average or typical system performance has impaired system predictability forcing analysis to resort to more conservative methods (i.e., methods that overestimate the real worst case). While architectures with higher predictability have been proposed [29,51], design practice currently has to live with the ongoing trend.

In some respect, efficient formal timing verification is even harder than function verification because of systems integration. Today, a vehicle, an aircraft, a medical device, and even a smartphone integrates many applications sharing the same network, processors, and run-time environment. This leads to potential

timing interference of seemingly unrelated applications. The integrated modular architecture (IMA), standardized as ARINC 653 [45] for aircraft design, and even more the automotive AUTOSAR standard are perfect examples for such software architectures. They also stand for different philosophies. While ARINC 653 takes a constructive approach and uses scheduling to obtain application isolation at the cost of resource efficiency, AUTOSAR does not constructively prevent timing interference, but the related automotive safety standard ISO 26262 requires proof of "freedom from interference" for safety critical applications.

However, even with extensive runs on millions of cases, simulation and prototyping remain an investigation of collections of use cases with decreasing expressiveness for large integrated systems. Therefore, there is a strong incentive to use formal timing analysis methods at least on the network level. For example, there are formal methods for some protocols such as the automotive Controller Area Network (CAN) bus which is the dominant automotive bus standard today [10]. Unfortunately, current automotive systems are not only large but heterogeneous combining different protocols and scheduling and arbitration strategies. To make things worse, the component and network technology incrementally develops over time challenging flexibility and scalability of any formal timing analysis.

In this situation, the introduction of modular timing analysis methods which support composition of analyses for different scheduling and arbitration strategies as well as different architectures and application systems with a variety of models-of-computation was considered a breakthrough. Today, most automotive manufacturers and many suppliers use formal timing analysis as part of their network development. A corresponding tool, SymTA/S, has been commercialized and is widely used. The original ideas which led to that tool can be found in [37].

This chapter presents the general concept of the Compositional Performance Analysis (CPA) and extensions of the last couple of years. Since this is an overview chapter, it stays on the surface to keep readability. For more details, the reader is referred to the large body of related scientific papers covering compositional performance analysis and a related approach based on the Real-Time Calculus (RTC) [49].

23.2 Fundamentals

CPA is an analysis framework which serves to formally derive the timing behavior of embedded real-time systems.

From a hardware perspective, an embedded real-time system consists of a set of interconnected components. These components include communication and computation elements as well as sensors and actuators which act as the connection to the system environment. The interconnected components represent the platform on which software applications with real-time requirements are executed. A software application is composed of tasks, entities of computation, which are distributed over and executed on different components of the system.

The execution order of tasks belonging to one application is constrained, for instance, the read of a sensor must be performed before the computation of a control law and the control of an actuator. Moreover, if several tasks are executed on one component, the tasks have to share the processing service the component offers. This has obviously an impact on the timing behavior of each task. As a result, in order to determine the timing behavior of the system, it is not sufficient to focus on isolated tasks. Apart from the interaction of tasks which are caused by functional dependencies (imposed execution order), nonfunctional dependencies (share of component service) have also to be taken into consideration.

In the following, the system model used for CPA is described, and then the compositional analysis principle is deduced. Following the above argumentation, CPA structures its system model with respect to three aspects: (1) the individual tasks, (2) the individual components with intra-component (local) dependencies between mapped tasks, and (3) the system platform with inter-component (global) dependencies between mapped tasks. The analysis is structured according to the local and global aspects, and it is compositional in the sense that the timing properties of the system can be conclusively derived from its constituting components.

23.2.1 Timing Model

In the following the timing model of a real-time embedded system is described as it is used in CPA. The timing model is layered and includes the task timing model, the component timing model, and the system timing model. All three layers are explained in detail below and are illustrated in Fig. 23.1.

23.2.1.1 Task Timing Model

In this section the timing behavior of an individual task τ_i is presented which is characterized completely by its execution time C_i. The execution time C_i is the amount of service time which a component has to provide in order to process task τ_i. The actual execution time of a task τ_i does not only heavily depend on the task input and the task state but also on the execution platform. For instance, the processor architecture, the cache architecture, and the Direct Memory Access (DMA) policy impact the execution time. As a result, a task τ_i does not have a static but rather a varying execution time C_i as illustrated in Fig. 23.2. For the CPA, the lower and the upper bound on the task execution are of interest because they include every intermediate timing behavior. The lower bound on the task execution time is called Best-Case Execution Time (BCET) denoted as C_i^-, whereas the upper bound is called the WCET denoted as C_i^+.

Different methods exist to derive the BCET and WCET of a task τ_i. One method is to simulate the task execution under different scenarios and observe the required execution time. Since the number of test cases is naturally limited, the simulation is bound not to cover all corner cases thus underestimating the WCET and overestimating the BCET. Formal program analysis, on the other hand, evaluates the source or object code associated with a task τ_i and takes into account

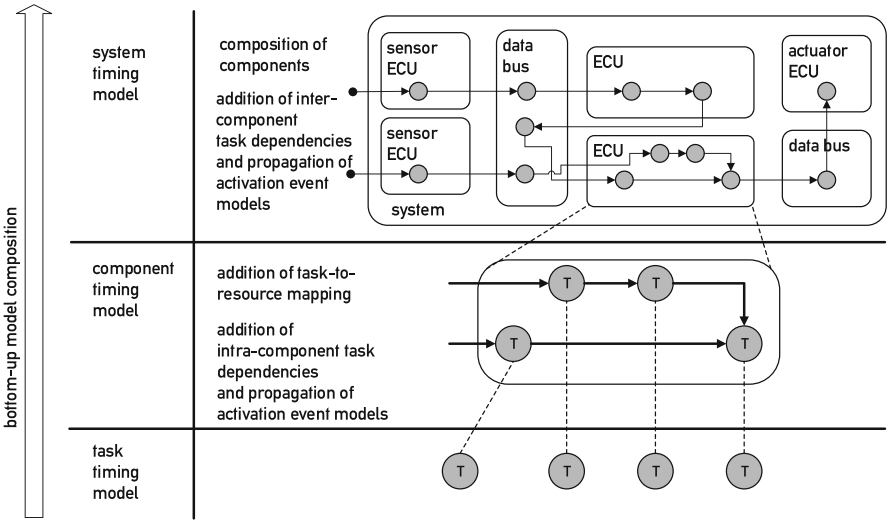


Fig. 23.1 System timing model. The system timing model used in CPA comprises three aspects: (1) individual tasks, (2) individual components with mapped tasks and local task dependencies, and (3) the entire platform with mapped tasks and global task dependencies

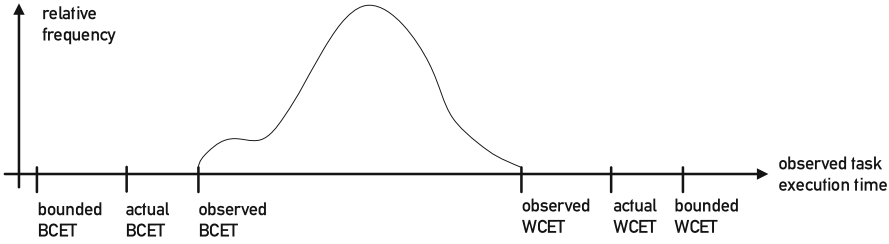


Fig. 23.2 Relative frequency of the observed execution time of a task. Observed execution times from simulations can in general not bound the actual best-case and worst-case behaviors. In contrast, a formal bound tends to overestimate the actual WCET and to underestimate the actual BCET

the architectural properties of the execution platform. Program analysis is capable of deriving a safe upper bound on the WCET of a task τ_i, relying on worst-case program configuration and worst-case program input in order to cover all behavioral corner cases. Apart from static WCET bounds, parametric WCET bounds have been proposed [9]. Tools for the derivation of WCET bounds like aiT [1] are available. As programs can be become arbitrarily complex, program analysis makes conservative assumptions leading to significant WCET overestimations and BCET underestimations. This is the reason why program analysis for BCET/WCET-estimation is often only used for tasks with high criticality, i.e., with high impact on system safety. While there is extensive research dedicated to the WCET analysis

[52], it is not part of CPA itself. Therefore, the BCETs and WCETs are assumed to be known as safe but possibly conservative values.

23.2.1.2 Local View: Component Timing Model

The component timing model represents a locally restricted view on the system. It focuses on the timing behavior in the scope of individual system components.

Once a set of tasks is mapped to a component, the timing behavior of each task can no longer be treated in isolation. On the one hand, tasks interact due to *nonfunctional dependencies*, i.e., the share of component service which is determined by the local scheduling policy. On the other hand, tasks interact due to *functional dependencies*. All tasks belonging to one application have activation patterns, and an execution order imposed by the application which per definition specifies the high-level functional context and the functional interaction of tasks.

The derivation of a task activation pattern and the realization of the required execution order of tasks is done by the propagation of activation events in the CPA component timing model. Assume a precedence-constrained execution order of a set of tasks which are mapped to a component as illustrated in Fig. 23.1 for the component timing model layer. The activation pattern of a task τ_i which represents the first element in the execution order of tasks (head of a task chain) is determined by the behavior of an event source outside of the component. Such an event source can either be located in the system environment or at another system component. A task τ_j which directly succeeds task τ_i in the task chain is activated by the termination events of task τ_i. This propagation of activation events applies for all elements in the task chains. Note that the event propagation in CPA timing model implies that the activation patterns of tasks are derived from high-level functional constraints, and do not represent a direct property of the individual task.

In this section, first the component abstraction and then the component scheduler is introduced which organizes the share of component service. Hereafter, in order to reason about service demand and event propagation among precedence-constrained tasks, the principles of activation/termination traces and activation/termination event models are explained.

Component Abstraction

System components are also called *resources* and are characterized by their property to provide processing service to tasks. A component can either be a *computation resource* or a *communication resource*, and depending on the kind of resource, a task can either represent an algorithm to be computed or a data frame to be transmitted. For instance, an Electronic Control Unit (ECU) is a computation resource which may provide processing service to tasks computing a control law. A data bus, in contrast, can be modeled as a communication resource transmitting data frames.

Scheduler

A resource can only serve one task at a time; hence, if more than one task is ready to execute, it has to be decided which task will be processed next. The decision process is performed by the *scheduler* of the resource. It specifies when to start and pause

the execution of pending tasks. Commonly used scheduling policies for embedded real-time systems are Static Priority Preemptive (SPP) and Static-Priority Non-Preemptive (SPNP) *scheduling*. We will use these policies as important examples throughout the chapter, noting that CPA is not limited to static priority policies.

Activation Traces and Termination Traces

A task is activated by an *activation event*, where activation means that the task is moved from a sleeping state to a ready-to-execute state. Such an activation event can either be *time triggered* or *event triggered*. A time-triggered activation occurs according to a predefined time pattern, whereas an event-triggered activation is a reaction to a certain true condition in the system or the system environment. Additionally to being activated by an activation event, a task produces a *termination event* when it has finished.

An *activation trace* of a task τ_i describes the set of instants at which an activation event for a task τ_i takes place. A similar definition applies to the *termination trace*. An activation event for a task τ_i originates either from an *event source* which is triggered by the system environment or another external event source like a timer, or it is produced by a predecessor task τ_j if a precedence constraint with respect to the execution order exists between two tasks in the form of $\tau_j \rightarrow \tau_i$ (τ_j precedes τ_i). The termination event of the predecessor task τ_j, produced at the end of its execution, then represents the activation event for task τ_i. If the predecessor task is not local to the component, an external event source with a conservative activation pattern is initially assumed in the CPA component timing model, see Sect. 23.2.2.2.

Activation event traces from event sources and predecessor tasks can be combined to form a joint activation event trace. The *join* of two event traces can either follow an AND or an OR logic. An AND logic implies that only if an event from both incoming event traces is available at a given time, an event is produced on the joint event trace. An OR logic requires an event from only one of the two incoming event traces to produce an event on the joint trace. It is also possible for a single event trace to *fork*, i.e., to serve as an activation trace for multiple tasks. Figure 23.3 illustrates how activations events propagate through the system. Note that activation and termination events do not include any information on data, rather the event concept is agnostic of the concrete processed and transferred data and focuses on the timing behavior of tasks.

Event Models

An event trace of a task τ_i captures the sequence of event occurrences with respect to task τ_i for a given system trajectory in the system state space. Due to the countless number of possible system trajectories, it is not feasible to perform a timing analysis for all possible event traces. Rather, the timing analysis has to be restricted to corner cases which bound the timing behavior of the task τ_i in all other cases. CPA uses for this purpose *arbitrary activation [termination] event models* where arbitrary means that an arbitrary activation [termination] behavior of a task τ_i can be bounded by the event model [36].

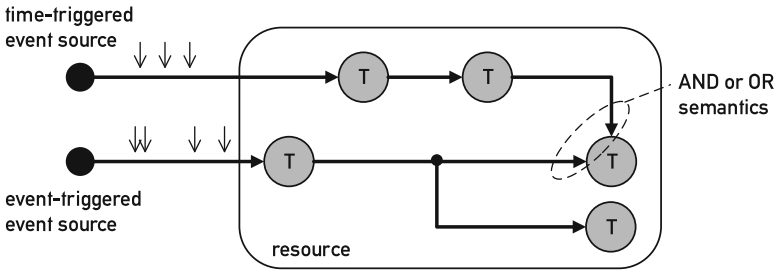


Fig. 23.3 Event traces. Tasks are activated by activation events from time-triggered or event-triggered event sources as well as by the termination events of predecessor tasks. The *arrows* connecting event sources and tasks as well as tasks among each other indicate the flow of events through the system. The *small arrows* ↓ indicate individual activation events which occur regularly in case of time-triggering and irregularly in case of event-triggering and propagate through the system

An event model of a task τ_i is defined by the set of two *distance functions* $\delta_i^-, \delta_i^+ : \mathbb{N}_0 \to \mathbb{R}_0^+$, namely, the minimum distance function δ_i^- and the maximum distance function δ_i^+. The minimum distance function $\delta_i^-(n)$ describes the minimum time distance between any n consecutive activation [termination] events of task τ_i. The maximum distance function $\delta_i^+(n)$ describes the maximum time distance between any n consecutive activation [termination] events of task τ_i. The distance between zero and one events is defined for mathematical convenience as zero so that $\delta_i^{+,-}(0) = \delta_i^{+,-}(1) := 0$.

Pseudo inverses of the distance functions are the *arrival functions* $\eta_i^+, \eta_i^- : \mathbb{R}_0^+ \to \mathbb{N}_0$. The maximum arrival function η_i^+ is the pseudo inverse of the minimum distance function δ_i^-, and the minimum arrival function η_i^- is the pseudo inverse of the maximum distance function δ_i^+. The function $\eta^+(\Delta t)$, resp. $\eta^-(\Delta t)$, returns the maximum, resp. minimum, number of activation [termination] events of task τ_i within any half-open time interval $[t, t + \Delta t)$. For a time interval $\Delta t = 0$, the event arrival functions η_i^- and η_i^+ are defined as zero. The pseudo inverses are introduced because they often allow a more elegant mathematical formulation of timing analysis problems.

The pair of minimum and maximum distance functions, resp. arrival functions, describe the best-case and worst-case event trace of a task τ_i with respect to event frequency. If distance functions, resp. arrival functions, cannot be formally derived, it is possible to extract them from measured event traces [17].

A commonly used event model is the *PJd event model* [36] shown in Fig. 23.4. A PJd event model is applicable to a task τ_i which is activated periodically but may experience bursts of activations once in a while. It can be characterized by the three parameters period T_i, jitter J_i, and a minimum event distance d_i. Bursts occur if the jitter of periodic activation events, i.e., the maximum relative deviation in time from the exactly periodic activation instant, is larger than the task activation period. In this case the task may receive multiple new activation events before the current task invocation has terminated. The PJd event model is also of historical

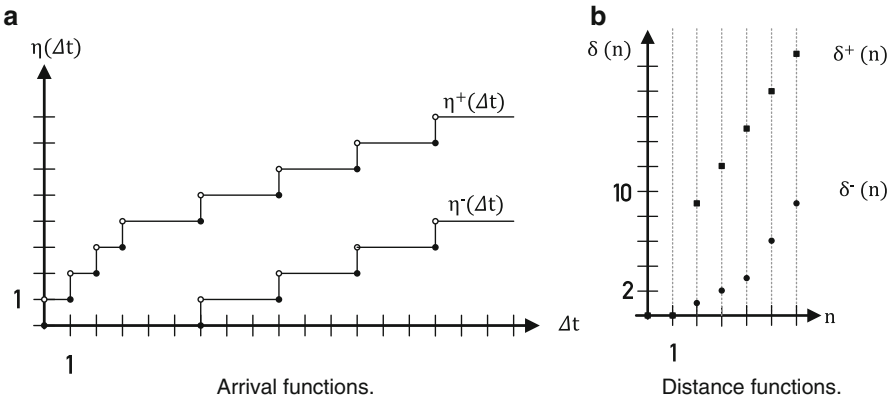


Fig. 23.4 PJd event model. (**a**) illustrates the maximum arrival function $\eta^+(\Delta t) = \min\left\{\left\lceil \frac{\Delta t}{d} \right\rceil, \left\lceil \frac{\Delta t + J}{T} \right\rceil\right\}$ and the minimum arrival function $\eta^-(\Delta t) = \max\left\{0, \left\lceil \frac{\Delta t - J}{T} \right\rceil\right\}$ for a PJd event model with $T = 3, J = 6, d = 1$. (**b**) illustrates for the same PJd model the minimum distance function $\delta^-(\Delta t) = \max\{(n-1)\cdot d, (n-1)\cdot T - J\}$ and the maximum distance function $\delta^+(\Delta t) = (n-1)\cdot T + J$ with $n \geq 2$. For small time intervals, the burst behavior dominates where the minimum event distance d bounds the maximum event frequency during the burst. For large time intervals, the periodic behavior with jitter dominates

importance because it was the basis for the description of activation patterns before the more general distance functions and arrival functions were introduced [18, 36]. It has the advantage that it can be described by a limited set of parameters and shows a periodic behavior for larger time intervals. In contrast, arbitrary event models may potentially extend indefinitely without showing a repetitive pattern. This complicating property of arbitrary event models has to be handled by an appropriate analysis approach which is able to extract relevant limited time windows for the investigation of the system timing behavior, see the busy period concept in Sect. 23.2.2.

23.2.1.3 Global View: System Timing Model

From a global perspective, the system is composed of a set of interconnected communication and computation resources which constitute the processing *platform*. Tasks are mapped to the resources and interact with each other forming larger functional entities, so-called applications.

In the global perspective, inter-component interactions between tasks as defined by applications become visible; see Fig. 23.1. Inter-component interactions between tasks are, like intra-component interactions, modeled by event propagation in CPA. To account for a dependency between two tasks τ_i, τ_j mapped on two different components where task τ_i precedes task τ_j, the termination event model of the predecessor task τ_i is propagated as activation event model to the successor task τ_j. The required termination event model of a task is determined iteratively during the system analysis procedure described in the following section.

23.2.2 Analysis

CPA is a systematic timing analysis method which serves to verify the timing properties of complex distributed real-time systems with heterogeneous components. The major challenge in analyzing such a system is to take into account the numerous interdependencies of task executions which result both from direct task interaction and indirect task interaction due to the share of resources.

CPA follows a compositional approach which first performs a local component-related timing verification step and then, in a global timing verification step, sets the local verification problems in a system context where inter-component dependencies are considered. The inter-component dependencies relate the local verification problems in such a manner that their inputs and outputs are linked. The relation of the local verification problems leads to a fixed point problem which converges if the propagation of outputs to inputs between related verification problems does not change the verification results any more. If the system is overloaded, the fixed point problem does not converge, and an abort criterion, e.g., the detected miss of a task deadline, is used to stop the iteration process.

In this section, first the local analysis is presented and then the superordinate global analysis is introduced.

23.2.2.1 Local Analysis

Local analysis refers to the analysis of timing properties of an individual system resource which processes tasks according to a given scheduling policy. The local analysis is based on the component timing model.

Resource Utilization

The *utilization* U of a resource is defined as the quotient of the execution request which the resource receives and the available service time which it can provide. It is computed by accumulating the utilization U_i that each task τ_i with $i = 1 \ldots N$ mapped to this resource imposes. The maximum utilization U_i^+ that an individual task τ_i can impose on a resource is given if the task requests its maximum execution time C_i^+ at its maximum activation frequency

$$U_i^+ = \lim_{n \to \infty} \frac{n \cdot C_i^+}{\delta_i^-(n)}. \tag{23.1}$$

The maximum utilization of a resource $U^+ = \sum_{i=1}^{N} U_i^+$ is an important variable to determine whether the resource is overloaded, and consequently the tasks are not schedulable. Apparently, it is impossible to schedule tasks sets with a resource utilization larger than one. In this case, the local analysis will be stopped. While being a necessary (under some conditions even sufficient) indicator for the schedulability of a task set, the utilization is not an appropriate means to describe the system timing behavior in detail. The utilization of a resource does not give any

insight into the sequence of execution and suspension phases during the processing of a task which are determined by the applied scheduling policy.

Worst-Case Response Time

Since tasks which are allocated to the same resource have to share its service, the processing of a task τ_i is preempted if other tasks with higher priority are activated. This is illustrated in Fig. 23.5. The time interval between the activation and the termination of a task τ_i, including all suspension phases, is defined as the response time of task τ_i denoted as R_i. The minimum response time of task τ_i, denoted as R_i^-, and the maximum response time of task τ_i, denoted as R_i^+, serve to bound the response time behavior of task τ_i.

The decisive property of hard real-time systems is that no task τ_i is allowed to miss its deadline D_i. In other words, it is required that the deadline D_i is an upper bound on the worst-case response time R_i^+. A major purpose of timing analysis like CPA is to verify this system property which is crucial for safe operation of real-time systems.

Determining the Worst-Case Response Time

In order to verify whether the real-time requirement $R_i^+ \leq D_i$ for a task τ_i is fulfilled, the worst-case response time R_i^+ needs to be determined. Obviously, it is not possible to explore the entire system state space for this purpose. Rather a worst-case scenario has to be derived which allows to find the worst-case response time R_i^+ in a limited time window.

The limited time window of system behavior comprising the worst-case response time behavior of task τ_i is called the *longest level-i busy period* [23]. The longest level-i busy period is initiated by the so-called critical instant. The critical instant describes the alignment of task activations and the execution times which lead to the maximum interference with respect to task τ_i and consequently to the worst-case response time R_i^+. The longest level-i busy period closes if the investigation of a longer time interval is known not to contribute any new information to the worst-case response time analysis. In the following, first the concept of the level-i busy period is explained. Then the multiple activation scheduling horizon as well as the multiple activation processing time processing time are introduced. Both of those

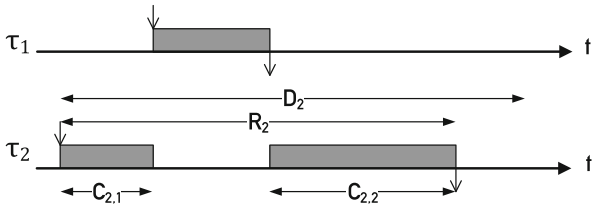


Fig. 23.5 Response time and relative deadline. Task τ_2 is preempted during execution by task τ_1 which is of higher priority. Therefore, the response time R_2 is larger than the execution time $C_2 = C_{2,1} + C_{2,2}$. The response time R_2 is still smaller than the deadline D_2

variables serve to formally describe the processing behavior of a local resource with respect to task τ_i within the level-i busy period.

Busy Period

A level-i busy period [23] is a time interval during which a resource R is busy processing a task τ_i or tasks of higher priority than task τ_i under a fixed priority scheduling policy. Directly before and after the level-i busy period, the resource R is idle with respect to task τ_i and tasks of higher priority.

The level-i busy period is an elegant means to perform a worst-case response time analysis by investigation of a limited time window. The idea is that the response time behavior of task τ_i in a level-i busy period is completely independent of events outside of this time interval. A level-i busy period is separated from the preceding and succeeding level-i busy periods by idle phases of the resource R. An idle phase implies that the resource R is virtually reset to an initial state being ignorant of previous execution requests of task τ_i or tasks of higher priority. It is thus sufficient to investigate in the timing analysis the level-i busy period which comprises the worst-case response time behavior of task τ_i. This so-called longest level-i busy period is initiated by the critical instant, which creates the maximum possible interference with respect to task τ_i so that the worst-case response time of task τ_i can be observed within the longest level-i busy period. It is a skillful task to derive this critical alignment of task activations and service requests for a given system configuration.

In the following, the longest level-i busy period and the initiating critical instant are derived for an SPP-scheduled task set on a single processing resource R with no restrictions on the task activation event models. Assume that at an instant $t - \epsilon$, the resource R is idle with respect to task τ_i and all tasks with higher priority. Shortly after at instant t, task τ_i is activated for the first time and requests its maximum execution time C_i^+. The activation causes the creation of the first task instance, also called job, which is denoted as $\tau_i(1)$. If all tasks with higher priority than task τ_i are activated simultaneously with $\tau_i(1)$ at t and request their maximum execution time at the highest possible frequency, then the maximum interference with respect to task $\tau_i(1)$ is evoked. This alignment of activations is the critical instant, the starting point of the longest level-i busy period. The level-i busy period generally comprises more than one job of task τ_i. The reason is that before job $\tau_i(1)$ terminates, a second activation of task τ_i may occur. Consequently the resource stays busy processing job $\tau_i(2)$ even if job $\tau_i(1)$ terminated, and incoming jobs of tasks with priority higher than task τ_i will preempt $\tau_i(2)$ from time to time. The same may of course happen before job $\tau_i(2)$ terminates etc., and only when a job of task τ_i finishes before a new activation for task τ_i comes in and no tasks with higher priority are processed, the level-i busy period closes.

The critical instant for a task τ_i scheduled under an SPNP policy occurs (1) if task τ_i is activated simultaneously with all tasks of higher priority, (2) if task τ_i and all tasks of higher priority request their maximum execution times at highest possible frequency, and (3) if a task τ_j of lower priority, which has the largest execution time

among all tasks with a priority lower than task τ_i, started execution just previously to the first activation of task τ_i.

In the formal response time analysis, the closure of the level-i busy period is represented by the solution of a fixed point problem. In order to be able to formulate a formal response time analysis, the multiple activation scheduling horizon and the multiple activation event processing time have to be introduced as done in the following paragraphs. Both variables mathematically describe the timing behavior of task τ_i within the level-i busy window.

Multiple Activation Scheduling Horizon

The *q-activation scheduling horizon* $S_i(q)$ of task τ_i is defined as the maximum half-open time interval which starts with the arrival of the first job $\tau_i(1)$ of any sequence of q consecutive jobs $\tau_i(1), \tau_i(2), \ldots \tau_i(q)$. The scheduling horizon closes at the (not included) point in time when a theoretical activation of task τ_i with an infinitesimally short execution time ϵ could be immediately served by the resource R after the processing of the q consecutive jobs. This theoretical activation is independent from the actual activation model of task τ_i since it is never actually executed [11].

The q-activation scheduling horizon generalizes the idea of the level-i busy period for a number q of activations. During the q-activation scheduling horizon, the resource R is busy processing task q jobs of τ_i and tasks of higher priority. The condition, which a theoretical activation with infinitesimally short execution time ϵ could be potentially served at the end of the scheduling horizon, enforces an idle time with respect to q jobs of task τ_i and all tasks of higher priority at the end of the scheduling horizon. The scheduling horizon for $q = q_i^+$ corresponds to the longest level-i busy period, where q_i^+ is the maximum number of activations of task τ_i which fall into the scheduling horizon of their respective predecessor jobs

$$q_i^+ = \min\{q \geq 1 \mid S(q) < \delta_i^-(q+1)\}. \tag{23.2}$$

For the SPP scheduling policy, the q-activation scheduling horizon $S_i(q)$ is the solution to the following fixed point equation

$$S_i(q) = q \cdot C_i^+ + \sum_{j \in hp(i)} C_j^+ \cdot \eta_j^+(S_i(q)). \tag{23.3}$$

As can be seen in Eq. 23.3, the scheduling horizon $S_i(q)$ is composed of two parts. Firstly, it contains the maximum time interval which is required to service q jobs of task τ_i. And secondly, it comprises the maximum interference caused by tasks of higher priority than task τ_i ($hp(i)$). The maximum interference is evoked if every task τ_j with $j \in hp(i)$ is activated according to its maximum arrival curve η_j^+ and every job requests the worst-case execution time C_j^+ during $S_i(q)$. At the end of $S_i(q)$, a hypothetical $q+1$st activation of task τ_i with ϵ execution time could immediately be served because all q jobs of task τ_i are processed and no jobs of

higher priority are pending. In case of the SPNP scheduling policy, the q-activation scheduling horizon $S_i(q)$ has to take into account the worst-case one-time blocking caused by a task of lower priority than task τ_i

$$S_i(q) = q \cdot C_i^+ + \max_{j \in lp(i)} \left\{ C_j^+ \right\} + \sum_{k \in hp(i)} \eta_k^+(S_i(q)) \cdot C_k^+. \tag{23.4}$$

Therefore, $S_i(q)$ is composed of (1) the maximum processing time of q jobs of task τ_i, (2) the maximum one-time blocking of task τ_i by a task of lower priority than task τ_i due to non-preemption, and (3) the maximum interference of tasks with a higher priority than task τ_i.

Multiple Activation Processing Time

The *q-activation processing time* $B_i(q)$ is defined as the time interval starting with the arrival of the first job $\tau_i(1)$ and ending at the termination of job $\tau_i(q)$ for any q consecutive activations of task τ_i which fall into the scheduling horizon of their respective predecessors.

The maximum q-activation processing time $B_i^+(q)$ serves as basis for the worst-case response computation. By definition, the maximum response time of the qth task instance, denoted as $R_i^+(q)$, is the difference of the maximum q-activation processing time of and its earliest possible time of activation

$$R_i^+(q) = B_i^+(q) - \delta_i^-(q). \tag{23.5}$$

The worst-case response time of a task τ_i, denoted as R_i^+, is the maximum response time of task τ_i within the longest level-i busy period, respectively, the q_i^+-activation scheduling horizon, thus

$$R_i^+ = \max_{1 \le q \le q_i^+} R_i^+(q). \tag{23.6}$$

For the SPP policy, the maximum q-activation processing time $B_i^+(q)$ is identical to the q-activation scheduling horizon $S_i(q)$ so that

$$B_i^+(q) = q \cdot C_i^+ + \sum_{j \in hp(i)} C_j^+ \cdot \eta_j^+(B_i^+(q)). \tag{23.7}$$

The identity of the q-activation processing time and the q-activation scheduling horizon is due to the sub-additive behavior of the SPP scheduling policy with respect to the processing times [11, 39]: $B_i^+(q+p) \le B_i^+(q) + B_i^+(p)$. This property is, however, not fulfilled for the SPNP scheduling policy. The maximum processing time $B_i^+(q)$ under the SPNP policy is the sum of the maximum queuing delay with respect to job $\tau_i(q)$, denoted as $Q_i^+(q)$, and the maximum execution time of job $\tau_i(q)$

$$B_i^+(q) = Q_i^+(q) + C_i^+. \tag{23.8}$$

The maximum queuing delay $Q_i^+(q)$ is the time a job $\tau_i(q)$ has to wait before it is selected for execution by the SPNP scheduler. Activations of tasks with a higher priority than task τ_i which occur during the execution of the job $\tau_i(q)$ do not prolong its processing time as by definition of the scheduling policy, it cannot be preempted once it has started executing. The queuing delay can be bounded from above by [10]

$$Q_i^+(q) = (q-1) \cdot C_i^+ + \max_{j \in lp(i)} \left\{ C_j^+ \right\} + \sum_{k \in hp(i)} C_k^+ \cdot \eta_k^+ \left(Q_i(q) + \epsilon \right). \tag{23.9}$$

The maximum queuing delay accounts for (1) the maximum execution demand of all jobs of task τ_i activated prior to job $\tau_i(q)$, (2) the longest one-time lower priority blocking due to non-preemption, and (3) the longest higher priority blocking during queuing. The infinitesimally long time interval ϵ added to the queuing delay serves to check whether another job interfering with task $\tau_i(q)$ could start exactly at the end of the iteratively computed queuing delay, thus it extends the investigated time window. Note that Eq. 23.8–23.9 and Eq. 23.4 are not identical since the fixed point iteration for $B_i^+(q)$ accumulates higher priority interference only during the queuing delay plus ϵ, whereas $S_i(q)$ takes also into account interference of higher priority during the execution of the qth job.

Example

Consider the timing diagram in Fig. 23.6 which illustrates the concept of the multiple activation scheduling horizon and the multiple activation processing time. The timing diagram shows three tasks τ_1, τ_2, and τ_3 which are scheduled on a common resource under an SPNP policy, where task τ_1 is of higher priority than task τ_2 and task τ_2 is of higher priority than task τ_3. The scenario represents the worst case with respect to the response time of task τ_2 since (1) task τ_3 is activated just prior to tasks τ_1 and τ_2 with maximum execution demand, (2) task τ_1 causes maximum interference of higher priority, and (3) task τ_2 always requests its maximum execution time at highest possible frequency.

In the timing diagram, the multiple activation scheduling horizons and the multiple activation processing times are indicated. All scheduling horizons and processing times start at time 0. The termination of job $\tau_2(1)$ marks the end of $B_2^+(1)$. The scheduling horizon $S_2(1)$ is longer than $B_2^+(1)$ due to the higher priority interference which prevents that a hypothetical activation of task τ_2 with an infinitesimally short execution time ϵ could be immediately served after the first job $\tau_2(1)$. Note that the scheduling horizon is defined as a half-open interval and thus does not take interference of higher priority into account which arrives exactly at the interval boundary of $S_2(1)$. Since the activation of $\tau_2(2)$ falls into the scheduling horizon $S_2(1)$ of job $\tau_2(1)$, $B_2^+(2)$ exists. Again, the scheduling horizon $S_2(2)$ is longer than the processing time $B_2^+(2)$, but the activation of job $\tau_2(3)$ does not fall into the scheduling horizon $S_2(2)$. Thus, $S_2(2)$ is the longest scheduling horizon and

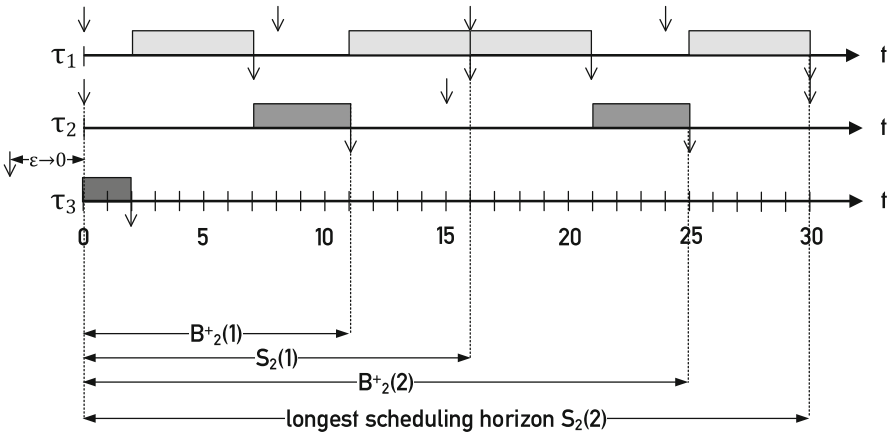


Fig. 23.6 Scheduling horizons and processing times for SPNP scheduling

represents the longest level-2 busy period. Note that the depicted scenario illustrates the non-subadditive behavior of the processing times: $B_2^+(2) > B_2^+(1) + B_2^+(1)$. The maximum processing time of job $\tau_2(1)$ is $B_2^+(1) = 11$, and its worst-case response time equals $R_2^+(1) = B_2^+(1) - \delta^-(1) = 11 - 0 = 11$. The maximum processing time of job $\tau_2(2)$ is $B_2^+(1) = 25$, and its worst-case response time equals $R_2^+(2) = B_2^+(2) - \delta^-(2) = 25 - 15 = 10$. Thus, a worst-case response time analysis yields the result $R_2^+ = \max_{1 \le q \le q_i^+} R_2^+(q) = 11$.

Best-Case Response Time

The best-case response time R_i^- and the worst-case response time R_i^+ serve to bound the response time behavior of task τ_i. A simple approximation of the best-case response time R_i^- relies on the following assumptions: (1) the absence of interference by tasks with equal or higher priority, and (2) the request of the minimum execution time C_i^-.

$$R_i^- = C_i^-. \tag{23.10}$$

Even though this approximation does not necessarily represent a tight bound, it is usually acceptable as timing analysis aims to provide real-time guarantees and thus focuses particularly on worst-case behavior.

Jitter

Jitter represents the maximum time interval by which the occurrence of a given event may deviate from the expected occurrence of the event. The response time jitter of a task τ_i can hence be calculated as the difference between the best-case response time R_i^+ and the worst-case response time R_i^-

$$J_{i,resp} = R_i^+ - R_i^-. \tag{23.11}$$

Backlog

It is possible that an activation event for a task τ_i arrives before the previously activated task instance has been processed, e.g., due to high interference or jitter. In this case, a *backlog* of activation events with respect to task τ_i arises. To prevent any loss of information, all activation events are queued until they are processed. The queue semantics in CPA is characterized by a First-In First-Out (FIFO) organization and a nondestructive write to and a destructive read from a queue storing activation events. The determination of an appropriate queue size for pending activation events of task τ_i is important both to avoid dropping of events and over-dimensioning. The maximum activation backlog for task τ_i, denoted as b_i^+, is bounded by

$$b_i^+ = \max_{1 \le q \le q_i^+} \left\{ 0,\ \eta_i^+(B_i^+(q) + o_{i,out}) - q + 1 \right\}. \tag{23.12}$$

The expression $\eta^+(\Delta t)$ represents the maximum possible number of task instances which can be activated in Δt, here with $\Delta t = B_i^+(q) + o_{i,out}$. The first term $B_i^+(q)$ is the maximum processing time for the task instance q of τ_i, the second term $o_{i,out}$ represents the maximum overhead required to remove a finished task instance from the activation queue [13]. The term $-q + 1$ accounts for the fact that previously activated task instances have already been finished. In other words, Eq. 23.12 computes the difference between the number of occurred activation events and processed ones, hence returning the number of pending activation events.

23.2.2.2 Global Analysis

In the previous section on the principle of the local analysis, we have shown how to compute the worst-case and best-case response times, the output jitter and the maximum activation backlog for a task with a given activation model. The local analysis is resource-related and does not consider any interaction between tasks on different resources. However, in reality many real-time applications are composed of multiple tasks which are distributed over several resources. For instance, a typical real-time application performs a control function which evaluates and processes sensor data in order to control an actuator according to a given control law. An exemplary mapping of such a real-time application to a platform with communication and computation elements is illustrated in Fig. 23.7.

In this section, it is shown how the global analysis integrates the dependencies of tasks on different resources into the analysis.

Consideration of Global Precedence Constraints

Tasks in an application can generally not be executed in an arbitrary order but have to be executed in a function-related order. The functional restrictions on the possible execution orders of tasks are expressed in form of *precedence constraints*. Precedence constraints can be described by a directed graph, the nodes representing the tasks, and the directed edges representing the directed execution dependencies. Paths in a precedence graph describe linear, branched, or even cyclic structures of dependencies.

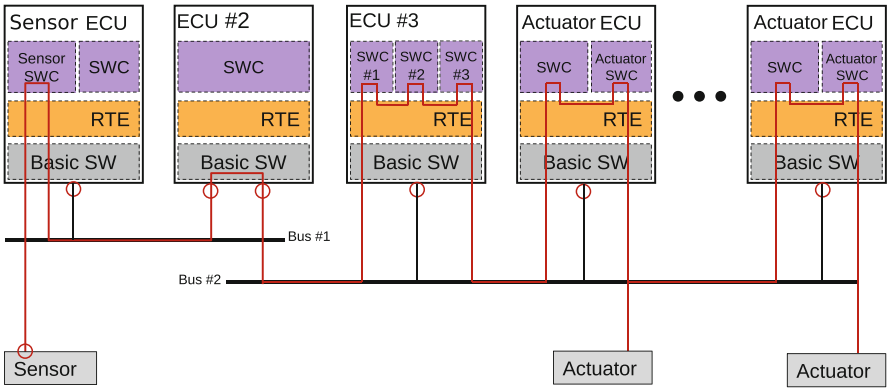


Fig. 23.7 Real-time application distributed over multiple computation and communication resources. ECU Electronic Control Unit, SWC Software Component, RTE Run-Time Environment

If two dependent tasks are mapped to two different resources, the timing behavior of those tasks is no longer exclusively determined by the local parameters. Consequently, the local view of a resource is no longer sufficient. To appropriately consider the precedence constraints of a pair of tasks where τ_i precedes τ_j ($\tau_i \rightarrow \tau_j$) in CPA, the termination event model of the predecessor task τ_i is used as the input event model of its successor task τ_j, i.e., the completion of one task triggers the activation of another. The termination behavior of task τ_i is bounded by the minimum output distance function, denoted as $\delta^-_{i,out}$, and the maximum output distance function, denoted as $\delta^+_{i,out}$. The distance functions naturally depend on the input (activation) event model ($\delta^-_{i,in}, \delta^+_{i,in}$) of task τ_i and the processing behavior of the resource [11, 36]

$$\delta^-_{i,out}(n) = \max\left\{\delta^-_{i,in} - J_{i,resp}, (n-1) \cdot d_{i,min}\right\} \tag{23.13}$$

$$\delta^+_{i,out}(n) = \delta^+_{i,in}(n) + J_{i,resp} \tag{23.14}$$

where $d_{i,min}$ is the minimum distance between any two terminations of task τ_i. A refined computation of the output models can be found in [43].

With the definition of inter-resource precedence constraints and the derivation of the best-case and worst-case output event models, the global analysis of the system can be performed.

Analysis Strategy

The global analysis is a timing verification step which sets the local verification problems in a system context where inter-resource precedence constraints are taken into account. As explained above, these inter-resource precedence constraints relate the local verification problems in such a manner that their input and output event models are linked. The relation of the local verification problems leads to a fixed

point problem which converges if the propagation of outputs to inputs between related verification problems does not change the verification results any more. Therefore, the CPA procedure consists of two parts: First, the local analysis of the individual resources is performed in order to generate the initial output event models. Then in a global analysis step, the output event models are propagated through the system to the tasks which utilize them as input event models due to global precedence constraints. The local analysis is repeated under updated input parameters computing best-case and worst-case response times, jitter and required queue sizes. Due to possible circular dependencies between activation models, it might be necessary to repeat this process multiple times. This propagation of output event models and update of input event models is continued until the analysis results converge.

In the following, the detailed procedure of the global analysis is presented which is also illustrated in Fig. 23.8:

1. For every task τ_i which is activated by events in the system environment, the input event model $(\delta^-_{i,in}, \delta^+_{i,in})$ is initialized with the input event model of the respective external event source.
2. For every task τ_j which is part of a precedence path, the input event model $(\delta^-_{j,in}, \delta^+_{j,in})$ corresponds to the output event model of the predecessor task. If in the initial analysis run no output event model of the predecessor task is available, then the input event model $(\delta^-_{j,in}, \delta^+_{j,in})$ is initialized with the input event model of the predecessor task.
3. A local analysis is performed for each resource with the objective of deriving the task output event models. Additionally, it is checked if the local analysis results violate any constraints, for instance, the required absence of system overload or the guarantee of all task deadlines.
4. The computed task output event models are propagated through the system to the tasks which utilize them as input event models due to respective precedence constraints.
5. If the propagated output event models are identical to the input event models used in the previous local analysis, a global fixed point has been reached and the analysis terminates [46]. All timing constraints, particularly task deadlines and end-to-end path latencies, are checked. The classical approach to compute the (worst case) end-to-end path latencies, is to accumulate the individual (worst case) response times for each task along the path [21, 39, 47]. If any constraint is violated, the system is not schedulable.

   Otherwise, if no fixed point has been reached yet, the local analysis is repeated with the updated input event models.

If the CPA has successfully terminated, the Best-Case Response Times (BCRTs) and Worst-Case Response Times (WCRTs) of each task are known such that the response time behavior of every task can be safely bounded. Moreover, maximum required queue sizes are derived. Further system performance results can be derived using the supplementary analysis modules of CPA presented in Sect. 23.3.

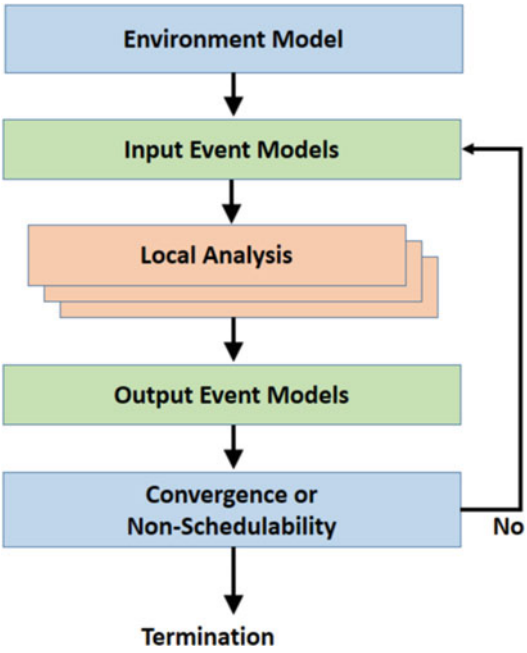


Fig. 23.8 In the system-level analysis, the local analyses are combined with a step to propagate updated output event models. This is repeated until the analysis converges or terminates due to abort criteria, e.g., constraint violation

23.3 Extensions

In the previous section, the basic CPA approach has been presented. It can be extended to cover more complex system models or to improve the analysis to compute tighter bounds for task response times or path latencies. In this section, CPA extensions are introduced which are able to deal with the usage of shared resources, the change of operation modi and the use of error handling protocols. Moreover an improved analysis for chained tasks is presented, and it is shown how CPA can be used to give formal timing guarantees for weakly-hard real-time systems.

23.3.1 Analysis of Systems with Shared Resources

In multi-core architectures several computational units have to share resources, such as the memory or data buses. For many of these shared resources concurrent or interrupted accesses are problematic, e.g., parallel write accesses to the same memory location can leave the accessed data corrupted or in an undefined state. Accesses to shared resources for which unconstrained accesses can be problematic are called *critical section*. Therefore, accesses to critical sections have to be isolated, i.e., locked by mutexes or semaphores. To ensure that only one process can have access to a critical section at a time, the accesses have to be modeled as

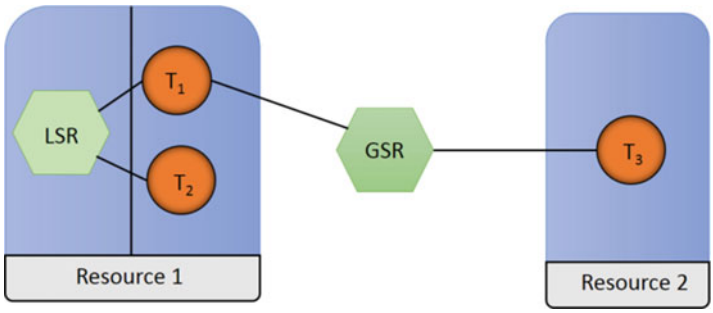


Fig. 23.9 Locally shared resources (LSR) can be only accessed by tasks from the same resource, globally shared resources (GSR) are available for any resource. As multiple tasks from different resources can attempt to access a GSR simultaneously, GSR require a different protection mechanism compared to LSR

non-preemptive sections [42]. If a task attempts to access a critical section, which has already been entered by another task, it has to wait until the current access has finished. This blocking time is further delaying the waiting task's response time as it cannot continue executing. Simply locking resources can lead to deadlocks or the priority inversion problem [22]. The automotive standard AUTOSAR, for instance, specifies different protocols for accessing locally shared resources (LSR) and globally shared resources (GSR) [3]. LSRs can only be accessed by tasks mapped on one computational resource, while GRSs can be accessed by any task in the system; see Fig. 23.9. Taking the different interference scenarios into account, AUTOSAR specifies to use the priority ceiling protocol for LSR and a spinlock mechanism for GSR [3].

When computing the WCRT of a task, the worst-case delay has to be assumed when accessing a shared resource. The scenario in which the worst-case delay occurs depends on the protocol used to protect the critical section. Generally, the worst-case delay occurs if τ_i tries to access a shared resource which is currently locked, and other tasks with a higher priority have accesses pending and continue to issue new ones. If the duration and number of accesses from all tasks is known, the maximum blocking a single access on a shared resource can be calculated [39]. Tasks often need to access a shared resource more than once. Calculating the independent worst-case delay for each individual access and accumulating the delay can provide an upper bound for the total blocking time a task can experience.

However, doing so can largely overestimate the actual possible interference, as the individual worst cases for each access are often mutually exclusive, i.e., blocking that can happen once is accounted for each single access.

To avoid this overestimation, the authors in [39] present a response time analysis combining arbitrarily activated tasks in multiprocessor systems with shared resources. Instead of the accumulation of the individual worst-case delays, they bounded the maximum possible interference that could occur during a given time interval, the task's processing time. The authors in [42] provided an analysis

framework to calculate task timing behavior under the multi-core priority ceiling protocol. In [28] the authors improved on this by taking into account local scheduling dependencies allowing to analyze sets of functional dependent tasks. The analysis has been extended to allow non-preemptive scheduling for tasks in multi-core architectures with shared resources in [25].

23.3.2 Analysis of Systems Undergoing Mode Changes

Some real-time systems have to execute in multiple different modi, being able to adapt to the environment or the mission being executed in multiple stages. A plane, for instance, has to operate differently during start, landing, or flight, while in a car the engine control might turn off certain analysis features, depending on the engine speed [26]. With the capability to change its configuration, the system is able to run more efficiently as it only needs to execute tasks when necessary and disables functions when no longer needed. Being able to deactivate unrequired functions can prevent or reduce expensive hardware over-dimensionation. Switching between different configurations is called a *mode change*. This comes with the requirement to analyze and verify the safety not only under one static configuration, but of the different operational modes and also of the transition phases. For real-time systems, this includes ensuring that the system satisfies all its deadlines during each possible configuration.

A system is running in a *steady state* if it is executing in one mode without any residing influence from a previous mode change, only executing the tasks from the corresponding task set. If the system receives a Mode Change Request (MCR) the set of running tasks has to be changed from the current mode to the new one. In order to evaluate the transition phase, each task is classified according to the following categories:

1. *Old tasks* are tasks which were present in the previous mode but not in the new one. They are immediately terminated when the MCR occurs, i.e., any active or pending task instances are removed from the system.
2. *Finished or completed tasks* are tasks which were present in the previous mode but not in the new one. During the transition phase, these tasks are allowed to finish their active and pending task instances, but no new instances will be started.
3. *New or added tasks* are tasks which are present in the new mode, but not in the old one. They can represent updated tasks from the old mode, e.g., with changed execution time or activation pattern, or new functionalities.
4. *Unchanged tasks* are present during the old and new modes with identical properties, only in systems with periodicity.

If the system's mode change protocol allows *unchanged tasks*, it is referred to as *with periodicity*, and *without periodicity* if all task sets are disjunct.

When a MCR occurs, the system has to change from the current mode to the new one, remove *old* and *finished tasks* and add *new tasks*. If the system waits until all *finished tasks* have completed their execution before starting to schedule *new tasks*, the

mode change protocol is called *synchronous*. Respectively, it is called *asynchronous*, if it starts scheduling *new tasks* right after the MCR has occurred, simultaneously with the last instances of *finished tasks*. Synchronous protocols ensure isolation between the modes and therefore do not require specific schedulability analyses for the transition phase. However, due to the delay introduced by the separation of the modes, synchronous protocols are not always feasible, if the transition has to be performed as fast as possible [27]. Asynchronous protocols, on the other hand, overcome this limitation and allow the simultaneous scheduling of tasks from the old and new mode. With an asynchronous protocol, the *new tasks* are added to the set of scheduled tasks, hence possibly increasing the resource utilization. As simply adding new tasks to the current set can lead to temporal overload on the resource, asynchronous protocols require specific schedulability analyses [27, 34, 35, 50]. For the remainder of this section, we will focus be on asynchronous protocols.

The CPA approach provides functionality to analyze the WCRTs of tasks and path latencies for a system in a certain mode. The transition periods can be modeled conservatively, by assuming that all tasks from the two (previous and new) modes are active simultaneously. In order to be able to model a transition phase, rules have to be defined regarding which transition phases can occur. The CPA extension relies on the assumption that the system executes in a steady state when the MCR occurs, i.e., new mode changes are not allowed to arise, while an older MCR is still exerting influence on the task activations. With this restriction, only tasks from exactly two mode sets need to be considered for a transition phase. The outcome from this is that not only the response times of tasks during steady states and the mode change phases are relevant but also the duration of these transition times. These *transient latencies* determine the distance to the next possible mode change [27].

The authors in [27] have shown that due to complex task dependencies the effects of a MCR are propagated delayed through the system, possibly causing feedback to the source of the MCR. They have shown how to bound the transition latency, by dividing it into local task transition latencies and global system transition latencies. In [26] the authors evaluate options for the design of multi-core real-time systems to minimize the impact of overly pessimistic measures taken in current practice.

23.3.3 Analysis of the Timing Impact of Errors and Error Handling

Safety-critical computing systems are usually designed to be fault tolerant toward communication and/or computation errors. However, each fault tolerance mechanisms incurs some time penalty because errors need first to be detected, and then an error correction or error masking measure has to be taken. To guarantee the correct timing behavior of a fault-tolerant safety-critical computing system, a formal performance analysis has to take these error-related, additional timing effects into account.

The consideration of timing overhead of fault tolerance mechanisms requires the adaptation of the local CPA. The stochastic nature of errors involves the introduction of a stochastic busy period; this is described in Sect. 23.3.3.2. Apart from timing

overhead fault-tolerant systems have specific precedence constraints which result, for instance, from redundant task executions. In Sect. 23.3.3.1, an adapted worst-case response time analysis is briefly outlined for such fault-tolerant systems.

23.3.3.1 Computation Errors and Error Handling

A basic principle of detecting computational errors is to perform the same computation several times and to compare the results. A discrepancy in the computed results is an evidence for an occurred fault. In a multi- or many-core processing system, it is possible to parallelize the redundant computations or, respectively, the execution of the redundant tasks. Such a fault tolerance approach leads to fork-join task graphs, where *forking* means the parallel execution of replicated tasks and *joining* means synchronization and the comparison of results. In [5] a strategy is presented how to derive worst-case response times for tasks in a task set with fork-join precedence constraints, so that the timing impact of replicated and parallelized computations can be evaluated.

23.3.3.2 Communication Errors and Error Handling

Unreliable communication links in data buses or packet-switched networks introduce bit errors in the digitally transmitted information. The occurrence of bit errors can be modeled by stochastic processes which often use the average bit error rate or packet error rate as an important parameter. If the assumption of independent bit errors is justified, the Bernoulli process or its approximation as a Poisson process is a classical modeling choice. If the probability of a bit error depends on past events, a state-aware Markov process is more appropriate [44].

For multi-master data buses and point-to-point communication, the detection of bit errors in transmitted frames at the receiver is typically based on error detecting codes. If a an error has been detected and signaled, a retransmission of the the corrupted or lost frame is initiated and the system is set back to a consistent state. Since bit errors can occur arbitrarily often albeit with a very low probability, the computation of a worst-case response time which includes an excessive detection, signaling, and correction time overhead is meaningless. A probabilistic scheduling guarantee, however, in the form of an exceedance function which specifies an upper bound on the probability that a task instance exceeds a reference response time value, is far more expressive. In [6], a probabilistic scheduling analysis is presented for a fault-tolerant multi-master/point-to-point communication system with non-preemptive fixed priority arbitration which is, for instance, applicable to CAN. The analysis computes first the worst-case response time of a frame under $1 \ldots K$ errors and the corresponding probabilities, and then derives an exceedance function by summing up the probabilities for all error scenarios which have a worst-case response time smaller or equal than the reference response time. In [4], an improved approach is presented which relies on stochastic frame transmission times. A stochastic frame transmission time is composed of the error-free frame transmission time and the stochastic overhead for error signaling and correction.

Stochastic frame transmission times give rise to stochastic busy periods from which stochastic response times and a less pessimistic exceedance function can be derived.

The performance analysis of switched real-time networks, both on-chip and off-chip networks, is treated in [7]. The network switches are assumed to employ a fixed priority-based arbitration scheme, and an end-to-end error protocol in form of an Automatic Repeat Request (ARQ) scheme is investigated. In the ARQ scheme, the sender buffers a sent packet until an Acknowledgement (ACK) message is received. If no ACK arrives at the sender in a given time interval, a timeout occurs and the buffered packet is retransmitted. The detection of corrupted packets at the receiver is typically based on error detecting codes. Both corrupted and lost packets are signaled to the sender by an omitted acknowledgement so that a retransmission is implicitly triggered. Variants of this type of ARQ error handling protocol are selective ARQ, stop-and-wait ARQ, and Go-back-N ARQ.

23.3.4 Refined Analysis of Task Chains

Real-time applications are usually not implemented as single tasks, but rather as a set of logical dependent tasks, as shown in Fig. 23.7. The tasks within an application are typically ordered and presented as a Directed Acyclic Graph (DAG), representing the logical order of execution. Within such a graph any logical dependent tasks form a *task path* or *task chain*. Sensor-actuator chains in automotive or avionic systems, for example, are distributed within the system as the components are physically apart, or information needs to be gathered in a central instance to perform decision-making. Multimedia applications on the other hand are often pipelined in order to process media streams more efficiently. To be able to take advantage of the parallelization of applications, the analyses methods need to support task paths and provide mechanisms to efficiently analyze the dependencies between tasks.

The basic CPA approach supports the latency analysis of task paths, as described in Sect. 23.2.2. The conservative approach is to compute the WCRT of each task which is an element of the considered task path and to derive the path latency by accumulating the individual WCRTs [21, 47]. While this simple accumulation provides an upper bound for the path latency, it is pessimistic if local worst-case scenarios within the same path are mutually exclusive.

In [38] the authors consider the communication between application threads and the corresponding precedence constraints in the resulting task graphs in order to improve the local WCRTs. They exclude infeasible worst-case scenarios for logically dependent tasks on the same SPP-scheduled resource by extending the scope of the busy period approach. By leveraging the particular semantics – including the distinction of synchronous and asynchronous communication – they were able to significantly reduce pessimism and the analysis complexity, resulting in a faster execution of the local analysis.

Another situation, which can lead to especially large local WCRTs, occurs when a task is activated with a burst. Bursts can potentially occur anywhere within the

system, but the same burst cannot occur on all resources at once. Accumulating the local WCRTs from a path with a bursty activation event model can therefore lead to a significant overestimation. Pessimistically bounded WCRTs can translate into over-dimensioning of the required hardware components and hence increased costs [39,41]. This issue is captured in the 'pay burst only once' problem [15].

In order to reduce the impact of the pay burst only once problem, the authors in [40] proposed a method to identify relevant combinations of local response times to derive a tighter worst-case path latency. They provided a methodology for computing path latencies, considering pipelined chains of tasks with transient overload along the path. This approach was extended in [41] by enabling the analysis of a wider variety of system topologies and including functional cycles and nonfunctional dependencies.

A similar method can be used to improve the analysis of Ethernet networks. In Ethernet networks, different data streams often need to be transferred with the same priority, as Ethernet switches only have a limited range of priorities. Streams with the same priority are queued and transferred according to FIFO scheduling. Therefore, for the individual worst-case analyses, each other stream with the same priority has be assumed to arrive simultaneously with the analyzed one but to be served first [13].

In [48] the authors have shown how the analysis of Ethernet networks can be improved by limiting the interference of tasks with the same priority that share more than one consecutive switch; see Fig. 23.10. Streams with the same priority that arrive simultaneously on one switch cannot arrive simultaneously on the following switch, as only one stream can be transfer at a time. This dependency can be exploited to provide tighter bounds for end-to-end path delays.

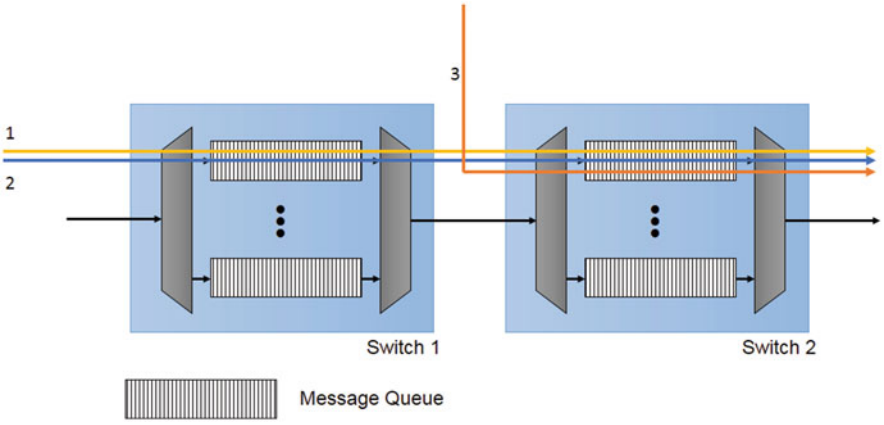


Fig. 23.10 The analysis of Ethernet networks can be improved, if dependencies between streams with the same priority – here stream 1 and 2 – are exploited. This can be done, if these streams are lead through the same two consecutive switches

23.3.5 Timing Verification of Weakly-Hard Real-Time Systems

Hard real-time computing systems require per definition that each instance of a task meets its deadline, whereas weakly-hard real-time systems tolerate occasional but in number and distribution precisely bounded deadline misses of tasks [8]. For instance, a weakly-hard system may require that a given task misses not more than m deadlines in any sequence of k consecutive task activations. The tolerance toward occasional deadline misses of tasks is usually based on the characteristics of the implemented real-time applications. Prevalent real-time applications like control functions, monitoring functions, and multimedia functions have shown to be robust against occasional but bounded sample or frame losses which can be interpreted as consequences of missed task deadlines. This robustness allows real-time applications to continue in safe operation even in the presence of limited transient overload [8, 16]. The analysis of weakly-hard real-time systems which is implemented as an extension of CPA, called Typical Worst-Case Analysis (TWCA) [2,31,32,53], provides formal guarantees for the compliance with weakly-hard real-time requirements for a wide range of system configurations. It scales to real-sized systems and provides a good computational efficiency [30].

TWCA assumes that each task has a typical behavior, e.g., a periodic activation pattern, which is captured in a typical activation model. In rare circumstances, a task may additionally experience nontypical activations, e.g., sporadic activations [31] or sporadic bursts [32], and then can be described by its worst-case activation model. The distance between the typical and the worst-case activation model of a task is captured by the so-called overload model. In the *typical worst case*, which occurs if all tasks show their densest pattern of typical activations and demand their maximum execution time, no deadlines are missed. In the *worst case*, which occurs if all tasks show their densest pattern of typical and nontypical activations and demand their maximum execution time, deadlines will be missed due to overload.

In two classical CPAs, the worst-case response times for both the typical worst case and the worst-case behavior of the system are computed: *Typical Worst-Case Response Time (TWCRT)* and *WCRT* for all tasks. If the WCRT of a task τ_i exceeds its deadline, a *deadline miss model* is computed which indicates the maximum number of observable deadline misses m in any sequence of k of consecutive instances of task τ_i. The computation of the deadline miss model relies on three main impact factors which need to be derived for each task interfering with task τ_i. Firstly, the overload model which is an indicator for how often nontypical activations can be encountered in a given time interval. Secondly, the longest time interval during which overload activations can impact the behavior of a sequence of k of consecutive instances of task τ_i. And thirdly, the maximum number of deadline misses of task τ_i that can be traced back to one overload activation. The computed deadline miss model for a task τ_i can be tightened if the number and distribution of overload activations, which induce the maximum number of deadline misses of task τ_i in any sequence of k consecutive instances, are bounded as precisely as possible [53].

23.3.6 Further Contributions

This chapter could only introduce the main functions of CPA. There are many more contributions that exceed the available space and should only be mentioned.

The robustness of a system, for instance, determines how sensitive the system reacts to changes in, e.g., execution and transmission delays, input data rates, or CPU clock cycles. A sensitivity analysis determines the influence of input data, or system configurations on the system robustness. The authors of [19, 20, 33] have shown how to identify critical components for the system robustness and how to optimize the platform. In many embedded systems, such as automotive systems, sensors are measuring the system behavior with a set period. If data is accessed periodically, but the communication path, e.g., a FlexRay bus, is transmitting the data with a different period, additional delay can occur due to the period mismatch. In [14] the authors discuss different end-to-end timing scenarios with a focus on register-based communication, taking different aspects of end-to-end delays into account.

23.4 Conclusion

In this chapter the compositional performance analysis approach has been presented. CPA provides a scalable framework to perform timing analysis of distributed embedded systems. It is widely used in the industrial development processes of real-time systems, especially in the automotive field where it is extensively proven in practice but also in avionics and even in networks-on-chip [12]. Numerous extensions exist to cover more complex applications, different applications of timing analysis in sensitivity, and robustness as well as error analysis.

Acknowledgments The project leading to this overview has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 644080 as well as from the German Research Foundation (DFG) under the contract number TWCA ER168/30-1.

References

1. AbsInt. aiT. http://www.absint.com/ait/. Accessed 24 Feb 2016
2. Ahrendts L, Hammadeh ZAH, Ernst R (2016) Guarantees for runnable entities with heterogeneous real-time requirements (to appear). In: Design, automation & test in Europe conference & exhibition (DATE 2016)
3. Autosar (2011) Specification of operating system, 5.0.0 edn. http://autosar.org/download/R4.0/AUTOSAR_SWS_OS.pdf
4. Axer P, Ernst R (2013) Stochastic response-time guarantee for non-preemptive, fixed-priority scheduling under errors. In: 50th ACM/EDAC/IEEE design automation conference (DAC 2013), pp 1–7. doi:10.1145/2463209.2488946
5. Axer P, Quinton S, Neukirchner M, Ernst R, Dobel B, Hartig H (2013) Response-time analysis of parallel fork-join workloads with real-time constraints. In: 25th Euromicro conference on real-time systems (ECRTS 2013), pp 215–224. doi:10.1109/ECRTS.2013.31

6. Axer P, Sebastian M, Ernst R (2012) Probabilistic response time bound for CAN messages with arbitrary deadlines. In: Design, automation test in Europe conference exhibition (DATE 2012), pp 1114–1117. doi:10.1109/DATE.2012.6176662
7. Axer P, Thiele D, Ernst R (2014) Formal timing analysis of automatic repeat request for switched real-time networks. In: 9th IEEE international symposium on industrial embedded systems (SIES 2014), pp 78–87. doi:10.1109/SIES.2014.6871191
8. Bernat G, Burns A, Liamosi A (2001) Weakly hard real-time systems. IEEE Trans Comput 50(4):308–321. doi:10.1109/12.919277
9. Bygde S (2010) Static WCET analysis based on abstract interpretation and counting of elements. Mälardalen University, Västerås
10. Davis RI, Burns A, Bril RJ, Lukkien JJ (2007) Controller area network (CAN) schedulability analysis: refuted, revisited and revised. Real-Time Syst 35(3):239–272. doi:10.1007/s11241-007-9012-7
11. Diemer J (2016) Predictable architecture and performance analysis for general-purpose networks-on-chip. Technische Universität Braunschweig, Braunschweig
12. Diemer J, Ernst R (2010) Back suction: service guarantees for latency-sensitive on-chip networks. In: Proceedings of the 2010 fourth ACM/IEEE international symposium on networks-on-chip (NOCS 2010). IEEE Computer Society, Washington, DC, pp 155–162. doi:10.1109/NOCS.2010.38
13. Diemer J, Rox J, Ernst R, Chen F, Kremer KT, Richter K (2012) Exploring the worst-case timing of ethernet AVB for industrial applications. In: Proceedings of the 38th annual conference of the IEEE industrial electronics society, Montreal. http://dx.doi.org/10.1109/IECON.2012.6389389
14. Feiertag N, Richter K, Nordlander J, Jonsson J (2008) A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In: Proceedings of the IEEE real-time system symposium – workshop on compositional theory and technology for real-time embedded systems, Barcelona, 30 Nov 2008
15. Fidler M (2003) Extending the network Calculus Pay bursts only once principle to aggregate scheduling. In: Proceedings of the quality of service in multiservice IP networks: second international workshop, QoS-IP 2003 Milano, 24–26 Feb 2003. Springer, Berlin/Heidelberg, pp 19–34. doi:10.1007/3-540-36480-3-2
16. Frehse G, Hamann A, Quinton S, Wöhrle M (2014) Formal analysis of timing effects on closed-loop properties of control software. In: 35th IEEE real-time systems symposium 2014 (RTSS), Rome. https://hal.inria.fr/hal-01097622
17. GmbH S. SymTA/S and traceanalyzer. https://www.symtavision.com/products/symtas-traceanalyzer/. Accessed 29 Jan 2016
18. Gresser K (1993) An event model for deadline verification of hard real-time systems. In: Proceedings of the fifth Euromicro workshop on real-time systems, 1993, pp 118–123. doi:10.1109/EMWRT.1993.639067
19. Hamann A, Racu R, Ernst R (2006) A formal approach to robustness maximization of complex heterogeneous embedded systems. In: Proceedings of the international conference on hardware – software codesign and system synthesis (CODES), Seoul
20. Hamann A, Racu R, Ernst R (2007) Multidimensional robustness optimization in heterogeneous distributed embedded systems. In: Proceedings of the 13th IEEE real-time and embedded technology and applications symposium
21. Henia R, Hamann A, Jersak M, Racu R, Richter K, Ernst R (2005) System level performance analysis – the symTA/S approach. IEE Proc Comput Digit Tech 152(2):148–166. doi:10.1049/ip-cdt:20045088
22. Lampson BW, Redell DD (1980) Experience with processes and monitors in mesa. Commun ACM 23(2):105–117. doi:10.1145/358818.358824
23. Lehoczky JP (1990) Fixed priority scheduling of periodic task sets with arbitrary deadlines. In: Proceedings of the 11th real-time systems symposium, pp 201–209. doi:10.1109/REAL.1990.128748
24. Liu JW (2000) Real-time systems. Prentice Hall, Englewood Cliffs

25. Negrean M, Ernst R (2012) Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources. In: Proceedings of the 7th IEEE international symposium on industrial embedded systems (SIES), Karlsruhe
26. Negrean M, Ernst R, Schliecker S (2012) Mastering timing challenges for the design of multi-mode applications on multi-core real-time embedded systems. In: 6th international congress on embedded real-time software and systems (ERTS), Toulouse
27. Negrean M, Neukirchner M, Stein S, Schliecker S, Ernst R (2011) Bounding mode change transition latencies for multi-mode real-time distributed applications. In: 16th IEEE international conference on emerging technologies and factory automation (ETFA 2011), Toulouse. http://dx.doi.org/10.1109/ETFA.2011.6059009
28. Negrean M, Schliecker S, Ernst R (2009) Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In: Proceedings of the Design, Automation, and Test in Europe (DATE), Nice. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5090720
29. Pellizzoni R, Schranzhofer A, Chen JJ, Caccamo M, Thiele L (2010) Worst case delay analysis for memory interference in multicore systems. In: Design, automation test in Europe conference exhibition (DATE 2010), pp 741–746. doi:10.1109/DATE.2010.5456952
30. Quinton S, Ernst R, Bertrand D, Yomsi PM (2012) Challenges and new trends in probabilistic timing analysis. In: Design, automation & test in Europe conference & exhibition (DATE 2012), pp 810–815. doi:10.1109/DATE.2012.6176605
31. Quinton S, Hanke M, Ernst R (2012) Formal analysis of sporadic overload in real-time systems. In: Design, automation test in Europe conference exhibition (DATE), pp 515–520. doi:10.1109/DATE.2012.6176523
32. Quinton S, Negrean M, Ernst R (2013) Formal analysis of sporadic bursts in real-time systems. In: Design, automation test in Europe conference exhibition (DATE), pp 767–772. doi:10.7873/DATE.2013.163
33. Racu R, Hamann A, Ernst R (2008) Sensitivity analysis of complex embedded real-time systems. Real-Time Syst 39:31–72
34. Rafik Henia RE (2007) Scenario aware analysis for complex event models and distributed systems. In: Proceedings real-time systems symposium
35. Real J, Crespo A (2004) Mode change protocols for real-time systems: a survey and a new proposal. Real-Time Syst 26(2):161–197. doi:10.1023/B:TIME.0000016129.97430.c6
36. Richter K (2005) Compositional scheduling analysis using standard event models. Ph.D. thesis, TU Braunschweig, IDA
37. Richter K, Jersak M, Ernst R (2003) A formal approach to MpSoC performance verification. Computer 36(4):60–67
38. Schlatow J, Ernst R (2016) Response-time analysis for task chains in communicating threads. In: 22nd IEEE real-time embedded technology and applications symposium (RTAS 2016), Vienna
39. Schliecker S (2011) Performance analysis of multiprocessor real-time systems with shared resources. Ph.D. thesis, Technische Universität Braunschweig, Braunschweig. http://www.cuvillier.de/flycms/de/html/30/-UickI3zKPS76fkY=/Buchdetails.html
40. Schliecker S, Ernst R (2008) Compositional path latency computation with local busy times. Technical report IDA-08-01, Technical University Braunschweig, Braunschweig
41. Schliecker S, Ernst R (2009) A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In: Proceedings of the 7th international conference on hardware software codesign and system synthesis (CODES-ISSS). ACM, Grenoble. http://doi.acm.org/10.1145/1629435.1629494
42. Schliecker S, Negrean M, Ernst R (2009) Response time analysis in multicore ECUs with shared resources. IEEE Trans Ind Inf 5(4):402–413. http://ieee-ies.org/tii/issues/iit09_4.shtml
43. Schliecker S, Rox J, Ivers M, Ernst R (2008) Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis. ACM, pp 185–190

44. Sebastian M, Axer P, Ernst R (2011) Utilizing hidden Markov models for formal reliability analysis of real-time communication systems with errors. In: IEEE 17th Pacific Rim international symposium on dependable computing (PRDC 2011), pp 79–88. doi:10.1109/PRDC.2011.19
45. Service ASC. Arinc 600 series. http://store.aviation-ia.com/cf/store/catalog.cfm?prod_group_id=1&category_group_id=3. Accessed 16 Mar 2016
46. Stein S, Diemer J, Ivers M, Schliecker S, Ernst R (2008) On the convergence of the symta/s analysis. Technical report, TU Braunschweig, Braunschweig
47. Sun J, Liu JWS (1995) Bounding the end-to-end response time in multiprocessor real-time systems. In: Proceedings of the third workshop on parallel and distributed real-time systems, 1995, pp 91–98. doi:10.1109/WPDRTS.1995.470502
48. Thiele D, Axer P, Ernst R (2015) Improving formal timing analysis of switched ethernet by exploiting fifo scheduling. In: Design automation conference (DAC), San Francisco
49. Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. In: Proceedings of the IEEE international symposium on circuits and systems. Emerging technologies for the 21st century, 2000, pp 101–104. doi:10.1109/ISCAS.2000.858698
50. Tindell KW, Burns A, Wellings AJ (1992) Mode changes in priority preemptively scheduled systems. In: Real-time systems symposium, 1992, pp 100–109. doi:10.1109/REAL.1992.242672
51. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenstrom P (2008) The worst-case execution time problem – overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3):Art. 36
52. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström, P (2008) The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3):36:1–36:53. doi:10.1145/1347375.1347389
53. Xu W, Hammadeh Z, Quinton S, Kröller A, Ernst R (2015) Improved deadline miss models for real-time systems using typical worst-case analysis. In: 27th Euromicro conference on real-time systems (ECRTS), Lund

Networked Real-Time Embedded Systems 24

Haibo Zeng, Prachi Joshi, Daniel Thiele, Jonas Diemer, Philip Axer, Rolf Ernst, and Petru Eles

Abstract

This chapter gives an overview on various real-time communication protocols, from the Controller Area Network (CAN) that was standardized over twenty years ago but is still popular, to the FlexRay protocol that provides strong predictability and fault tolerance, to the more recent Ethernet-based networks. The design of these protocols including their messaging mechanisms was driven by diversified requirements on bandwidth, real-time predictability, reliability, cost, etc. The chapter provides three examples of real-time communication protocols: CAN as an example of event-triggered communication, FlexRay as

This work was done while Daniel Thiele was with Technische Universität Braunschweig

H. Zeng (✉) • P. Joshi
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA
e-mail: hbzeng@vt.edu; haibo.zeng@gmail.com; prachi@vt.edu

D. Thiele
Elektrobit Automotive GmbH, Erlangen, Germany
e-mail: daniel.thiele@elektrobit.com

J. Diemer
Symtavision, Braunschweig, Germany
e-mail: diemer@symtavision.com

P. Axer
NXP Semiconductors, Hamburg, Germany
e-mail: philip.axer@nxp.com

R. Ernst
Institute of Computer and Network Engineering, Technical University Braunschweig, Braunschweig, Germany
e-mail: ernst@ida.ing.tu-bs.de

P. Eles
Department of Computer and Information Science, Linköping University, Linköping, Sweden
e-mail: petru.eles@liu.se

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_25

a heterogeneous protocol supporting both time-triggered and event-triggered communications, and different incarnations of Ethernet that provide desired temporal guarantees.

Acronyms

ADAS	Advanced Driver Assistance System
AFDX	Avionics Full-Duplex Switched Ethernet
ARQ	Automatic Repeat Request
AVB	Audio/Video Bridging
CAN	Controller Area Network
CPA	Compositional Performance Analysis
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
ECU	Electronic Control Unit
ET	Event-Triggered
FIFO	First-In First-Out
ILP	Integer Linear Program
LIN	Local Interconnect Network
MAC	Media Access Control
MOST	Media Oriented Systems Transport
QoS	Quality of Service
SPNP	Static-Priority Non-Preemptive
TSN	Time-Sensitive Networking
TT-CAN	Time-Triggered CAN
TTEthernet	Time-Triggered Ethernet
TTP	Time-Triggered Protocol
TT	Time-Triggered

Contents

24.1 Introduction

The communication network has been a key enabling technology for the advancement of distributed embedded systems, as evidenced in application domains like automotive, avionics, and industrial automation. With ever-increasing contents (safety, driver assistance, infotainment, etc.) in such systems relying on electronics and software, the supporting architecture is often integrated and interconnected by a complex set of heterogeneous data networks. For example, a modern automobile contains up to 100 Electronic Control Units (ECUs) and several heterogeneous communication buses (such as Controller Area Network (CAN), FlexRay, Media Oriented Systems Transport (MOST), and Local Interconnect Network (LIN)), exchanging thousands of signals [60].

This chapter intends to provide an overview of the real-time communication networks that are mainly categorized according to their messaging mechanism (or link layer protocol): Event-Triggered (ET), Time-Triggered (TT), or the coexistence of the two. In the first category (ET), messages are produced and transmitted based on the significant events occurring in the network, for example, the successful transmission of other messages. This category includes CAN, switched Ethernet, Avionics Full-Duplex Switched Ethernet (AFDX), and Ethernet Audio/Video Bridging (AVB). In the second category (TT), messages are transmitted at predefined time slots. Hence, it is essential to establish a global notion of time among all communication nodes to avoid possible collision. Time-Triggered CAN (TT-CAN) and Time-Triggered Protocol (TTP) belong to this category. In the third category, both time- and event-triggered traffics exist on the same communication network. Examples in this category include FlexRay, Ethernet Time-Sensitive Networking (TSN), and Time-Triggered Ethernet (TTEthernet).

The messaging mechanism in the communication network has a large impact on the temporal performance guarantees it can provide. Specifically, real-time performance metrics on the communication network include message *latency* and *jitter*. Message latency (or response time) is defined as the time that the message is ready for transmission to the time it is successfully received. Meeting the worst-case latency requirement is of fundamental importance to validate the system behavior against message deadlines. Jitter is the difference between the worst-case and best-case latencies, a key metric to the control performance of the features [44].

For time-triggered communication, the bandwidth is assigned to messages according to a time-triggered pattern, and the transmission of a message is triggered exactly when the global time reaches certain time points. Hence, a deterministic

Table 24.1 Summary of real-time communication protocols

Property	CAN	FlexRay	TTP	TT-Ethernet	AFDX	Switched Ethernet	AVB	TSN
Bandwidth	1 Mbps	10 Mbps	25 Mbps	1 Gbps	100 Mbps	100 Mbps–1 Gbps		
Message size (bytes)	0–8	0–254	0–240	46–1500	64–1518	42–1542		
Channels	1	2	2	2	2	1	1	2+
Messaging	ET	TT + ET	TT	TT + ET	ET	ET	ET	ET/ET + TT
Composability	No	Mixed	Yes	Yes	Yes	Yes	Yes	Yes
Medium access	CSMA/CA	TDMA + FTDMA	TDMA	TDMA + SPNP	SPNP + BAG	SPNP	SPNP + CBS	TDMA + SPNP
CRC error detection	15-bit	11-bit Header + 24-bit Tailer	24-bit	32-bit	32-bit	32-bit	32-bit	32-bit

communication pattern on the bus is guaranteed. The message latency/jitter can easily be calculated by checking the predefined transmission slots, which is independent from the rest of the network. On the contrary, in event-triggered communication, network delay occurs due to conflicts on media access and queuing mechanisms in the network. Event-triggered communication can be further classified by the temporal guarantees it can provide. In CAN, the dynamic segment of FlexRay, and switched Ethernet the temporal bounds on latency and jitter can oftentimes be analyzed a priori with its priority-based scheduling. However, changes to one node have an impact on the temporal behavior of messages from other nodes in the network. This makes the system composition difficult. As a remedy, several event-triggered communication protocols provide guaranteed service (bounds for latency and jitter) to a message regardless of the rest of the network, given that the message is produced at a speed that does not exceed the predefined maximum rate. Examples include the bandwidth allocation gap in AFDX and the rate-constrained messages in TTEthernet.

Table 24.1 summarizes the main communication protocols that are currently being deployed or considered in the application domains of real-time embedded systems. (Acronyms in the table: CSMA/CA, carrier sense multiple access with collision avoidance; TDMA, time division multiple access; FTDMA, flexible TDMA; SPNP, static priority non-preemptive; BAG, bandwidth allocation gap; CBS, credit-based shaping.) They all provide some degrees of predictability on temporal behavior. Those that are more suitable for non-real-time application, such as CSMA/CD-based Ethernet, are left out of the comparison. Besides the messaging mechanism and composability, the table also compares their maximum bandwidth, message size, number of (redundant) channels, medium access policy, and error detection.

The rest of the chapter describes in detail three different communication protocols, focusing on their messaging mechanism and timing predictability: CAN as an

example of event-triggered communication, FlexRay as a heterogeneous protocol supporting both time-triggered and event-triggered communications, and different incarnations of Ethernet that provide desired temporal guarantees.

24.2 Event-Triggered Communication: Controller Area Network

Controller Area Network is a broadcast digital bus designed to operate at speeds from 20 kbit/s to 1 Mbit/s, standardized as ISO/DIS 11898 [36]. The transmission rate depends on the bus length and transceiver speed. CAN is an attractive solution for embedded control systems because of its low cost, light protocol management, the deterministic resolution of the contention, and the built-in features for error detection and retransmission. Today CAN networks are widely used in automotive applications, as well as in factory and plant controls, robotics, medical devices, and some avionics systems.

Since its standardization in the mid-1990s, it has attracted a significant amount of research from the real-time systems community. The CAN protocol adopts a collision detection and resolution scheme, where the message to be transmitted is chosen according to its identifier. When multiple nodes need to transmit over the same bus, the lowest identifier message is selected for transmission. This arbitration protocol allows encoding the message priority into the identifier field and implementing priority-based scheduling.

24.2.1 CAN Message Format and Bus Arbitration

The CAN protocol has the message format of Fig. 24.1, where the sizes of the fields are expressed in bits. Priorities are encoded in the message identifier field (ID), which is 11 bits wide for the standard format. The identifier is used not only for bus arbitration but also for describing the data content (identification of the message stream). The CAN protocol requires that all contending messages have a unique identifier (unique priority). The other fields are the start-of-frame (SOF) bit; the control field (CTL) containing information on the type of message; the data field (DATA) containing the actual data to be transmitted, up to a maximum of 8 bytes; the checksum (CRC) used to check the correctness of the message bits; the acknowledge field (ACK) used to acknowledge the reception; the end-of-frame (EOF) delimiter used to signal the end of the message; and the idle space or inter-frame bits (IF) used to separate one frame from the following one.

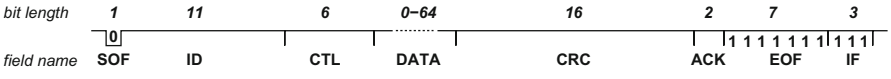


Fig. 24.1 The CAN data frame format

The CAN bus [36] essentially works as wired AND channels connecting all nodes. The two possible states encoded in the physical media are defined as "recessive" and "dominant," where dominant is a logical 0 (actively driven to a voltage by the transmitter) and recessive is a logical 1 (passively returned to a voltage by a resistor). The protocol allows multi-master access to the bus. At the lowest level, if multiple masters try to drive the bus state at the same time, the "dominant" configuration also prevails upon the "recessive."

The CAN arbitration protocol is both priority based and *non-preemptive*, as a message that is being transmitted cannot be preempted by higher-priority messages that are made available at the network adapters after the transmission has started. The media access protocol works by alternating contention and transmission phases. The time axis is divided into slots that must be larger than or equal to the time it takes for the signal to travel back and forth along the channel. The contention and transmission phases take place during the digital transmission of the frame bits.

At any time, if a node wishing to transmit finds the shared medium in an idle state, it waits for the end of the current bit (as defined by its internal oscillator) and then starts an arbitration phase by issuing a start-of-frame bit (a dominant bus state). At this point in time, each node with a message to be transmitted can start the competition to get access to the shared medium. All nodes will synchronize on the SOF bit edge and then, when the *identifier* field starts, they will serially transmit the identifier (priority) bits of the message they want to send, one bit for each slot, starting from the most significant ones. Collisions among identifier bits are resolved by the logical AND semantics, and each node reads the bus state while it transmits.

If a node reads its identifier bits on the medium without any change, it realizes it is the winner of the contention and is granted access to transmit the rest of the message. On the other hand, if one of the bits is changed when reading it back from the medium, this means that there is another node sending a higher-priority (dominant) bit; thus the message is withdrawn by the node that has lost the arbitration. The node stops transmitting the message bits and switches to listening mode only.

Clearly, according to this contention resolution protocol, the node with the lowest identifier is always the winner of the arbitration round and is transmitted next (the transmission stage is actually not even a distinct stage, given that the message frame is transmitted sequentially with the fields following the *Identifier*). All the other nodes switch to a listening mode. In a CAN system, at any time, there can be no two messages with the same identifier (this property is also referred to as *unique purity*). The message identifiers shall be assigned in a centralized fashion (e.g., by the system integrator). The arbitration is deterministic and also *priority* based, as that the message identifier gives in this case an indication of the message priority (the lowest identifier always wins).

24.2.2 Timing Analysis with Ideal Models

To predictably guarantee that the messages transmitting on CAN bus meet their deadlines, Tindell et al. [77] developed the seminal analysis of CAN message worst-case response times. The analysis is derived out of an analogy with CPU scheduling results but adapted to a message system scheduled by priority but without preemption. The result of the original paper influenced the design of on-chip CAN controllers like the Motorola msCAN and was included in the development of the early versions of Volcano's Network Architect tool. Volvo used these tools and the timing analysis results from Tindell's theory to evaluate communication latency in several car models, including the Volvo S80, XC90, S60, V50, and S40 [7]. The analysis was later found to be flawed by Davis et al. [12], where a set of formulas is provided for the exact evaluation and safe approximation of the worst-case message response times.

The analysis methods [12, 77] are based on a number of assumptions at the middleware, driver, and controller levels of the CAN communication stack, including

- The existence of a perfect priority-based software queue at each node for the outgoing messages
- The availability of one output buffer (i.e., transmit object, or simply TxObject) for each message, or preemptability of the TxObjects
- Immediate (zero-time) copy of the highest priority message from the software queue to the TxObjects

However, these idealized assumptions are often violated in practical systems (especially automotive systems) [15].

Under such analyses, each periodic or sporadic message m_i is defined by the tuple

$$m_i = \{id_i, T_i, J_i, C_i, D_i\}$$

where id_i is the CAN identifier, T_i is the period or the minimum inter-arrival time, J_i is the queuing jitter (sometimes also referred to as *release jitter*), C_i is the worst-case transmission time, and D_i is the deadline. The worst-case transmission time C_i is given by the total number of transmitted bits (including the worst-case stuffed bits) divided by the bus transmission rate.

The schedulability analysis [12] starts with the theorem that the worst-case response time is always inside the busy period. The busy period of priority level-i is a contiguous interval of time that starts at the critical instant. During the busy period, any message of priority lower than m_i is unable to start transmission and win arbitration. For m_i, the *critical instant* is the time instant where (1) the contention

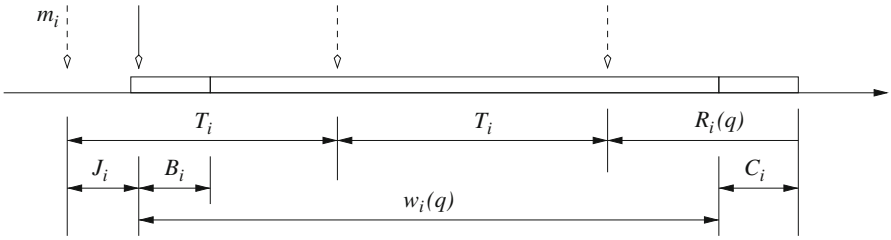


Fig. 24.2 The worst-case busy period and response time of message m_i

on the bus has just won by the longest lower-priority message (if one exists) and (2) all the higher-priority messages $hp(i)$ become simultaneously ready and arrive at their maximum rates thereafter.

Figure 24.2 illustrates the analysis of the worst-case response time for m_i. The dashed downward arrows denote the arrivals of m_i, separated by at least T_i. The solid downward arrow is the queuing time of m_i. The busy period consists of the queuing jitter, the blocking time, the queuing delay, and the transmission time of m_i itself.

To find the correct worst-case response time, the formula to be applied is a small modification to [77] that checks all the instances of message m_i transmitted inside the busy period. The response time of the q-th instance in the hyperperiod is

$$R_i(q) = J_i + w_i(q) - (q-1)T_i + C_i \tag{24.1}$$

where q ranges from 1 to the last instance $q_i^{\max}$ of m_i inside the busy period. The *blocking time*, i.e., the time spent on waiting for the transmission of a lower-priority message already on the bus when m_i becomes ready, is denoted as B_i. The worst-case queuing delay $w_i(q)$ for the q-th instance in the busy period is

$$w_i(q) = B_i + (q-1)C_i + \sum_{k \in hp(i)} \left\lceil \frac{w_i(q) + J_k + \tau_{bit}}{T_k} \right\rceil C_k \tag{24.2}$$

where τ_{bit} is the time to transmit one bit of data on the bus.

In Eq. (24.2), $w_i(q)$ appears on both sides. However, the right hand side is a monotonic nondecreasing function of $w_i(q)$. Hence, $w_i(q)$ can be solved using the iterative procedure defined in the equation below.

$$w_i^{n+1}(q) = B_i + (q-1)C_i + \sum_{k \in hp(i)} \left\lceil \frac{w_i^n(q) + J_k + \tau_{bit}}{T_k} \right\rceil C_k \tag{24.3}$$

The calculation can start with an initial value of $w_i^0(q) = B_i + (q-1)C_i$ and stop if $w_i^{n+1}(q) = w_i^n(q)$ or when $J_i + w_i^{n+1}(q) - (q-1)T_i + C_i > D_i$, the latter condition indicating that m_i is unschedulable.

The length of the longest busy period L_i and the index of the last instance are calculated as

$$\begin{cases} L_i = B_i + \sum_{k=hp(i)\bigcup\{i\}} \left\lceil \frac{L_i + J_k + \tau_{bit}}{T_k} \right\rceil C_k \\ q_i^{\max} = \left\lceil \frac{L_i + J_i}{T_i} \right\rceil \end{cases} \tag{24.4}$$

The worst-case message response time is the maximum among all its instances in the busy period.

$$R_i = \max_{q=1,\ldots,q_i^{\max}} \{R_i(q)\} \tag{24.5}$$

The above analysis (Eq. (24.1), (24.2), (24.3), (24.4), and (24.5)) requires to consider all the $q_i^{\max}$ instances in the busy period. Under the assumption that *the deadline is no greater than the period* ($D_i \leq T_i$), a sufficient but not necessary condition can be derived [12], which only checks the schedulability of the first instance by using its response time upper bound. Davis et al. [12] define the concept of *push through interference*, the largest interference that can be pushed through into the next period of message m_i due to the non-preemptive transmission of the previous instances. Any instance of m_i is subject to either direct blocking from lower-priority messages or indirect blocking from the previous instances of m_i, upper bounded by the push through interference of at most C_i. Hence, the queuing delay of the first instance ($q = 1$) of m_i is bounded by the result of the following equation:

$$w_i = \max(B_i, C_i) + \sum_{k\in hp(i)} \left\lceil \frac{w_i + J_k + \tau_{bit}}{T_k} \right\rceil C_k \tag{24.6}$$

which can be solved in a similar way to Eq. (24.2).

Besides the above analysis technique [12, 77], the schedulability of CAN messages is also addressed with other methods. Sufficient schedulability tests based on system utilization bound are derived for CAN (or in general, for systems under non-preemptive fixed-priority scheduling) with deadline monotonic and rate monotonic priority assignment policies [1, 6]. Navet et al. [62] introduce the concept of worst-case deadline failure probability because of transmission errors. In [5], Broster et al. present the probabilistic analysis of the impact of faults on CAN message latencies in the worst-case scenario. In [64], Nolte et al. extend the worst-case response time analysis using random message transmission times that take into account the probability of a given number of stuff bits. These works [5, 62, 64] still perform the analysis with respect to the *critical instant*, i.e., the worst-case response time scenario. Zeng et al. [81] provide a stochastic analysis framework for CAN message response times. In another work [82], they build a probability mix model to predict the distribution of message response times in CAN. The model is a mixture of the gamma distribution and the degenerate distribution. In [45], Liu et al. present an

extreme value theory-based statistical method to estimate the worst-case response times of CAN messages.

The analysis in [12, 77] has also been extended to other models of CAN messages. [8, 79] studied the worst-case response time for CAN messages that are scheduled with offsets. Mubeen et al. [51] consider the analysis for mixed messages, i.e., those with simultaneous time triggered (periodic) and event triggered (sporadic). They further provide analysis method to support mixed messages that are scheduled with offsets [55, 56]. In addition, the analysis on mixed messages is extended for nonideal CAN networks, including those where some nodes implement FIFO queues [54] or with CAN controllers implementing abortable [53] and non-abortable [52] transmit buffers. Liu et al. [46, 47] consider messages under the multi-frame task model [50] and its generalization [4] and present a sufficient schedulability analysis for systems with mixed message queues.

24.2.3 Analysis with Non-idealized Models

The analysis techniques in Sect. 24.2.2 assume an idealized model for the message queuing and the configuration and management of the peripheral TxObjects. In reality, CAN controllers have a limited number of available transmit buffers (TxObjects). When the number of TxObjects available at the controller is smaller than the number of messages sent by the node (as is the case for automotive gateways and nodes with lots of output information, or when message reception is polling based and a relatively large number of buffers must be reserved to input streams in order to avoid message loss by overwriting), it is necessary to use a *software queue* to hold messages waiting to be copied to a TxObject. Several commercial drivers (including those from Vector [78], probably the most commonly used in automotive systems) allow to put the outgoing messages in software queues as a temporary storage for accessing TxObjects. It is also quite common to use multiple queues, with each queue linked to a single TxObject. When a TxObject is available, a message is extracted from the queue and copied into it.

When software queues are used, the preservation of the priority order of the messages for accessing the CAN bus requires the following:

- **(a)** The messages in the software queue must be sorted by their priority (i.e., by message identifier).
- **(b)** When a TxObject (transmit buffer) becomes free, the highest priority message in the queue is immediately extracted and copied into the TxObject. In addition, messages in the TxObjects must be sent in the order of their CAN identifier.
- **(c)** If a higher-priority message becomes ready and all the TxObjects are used by lower-priority messages, the lowest-priority message in one of the TxObjects must be evicted and put back in the queue to ensure that a TxObject is available for the highest priority message.

When any of these conditions do not hold, priority inversion occurs, and *the worst-case timing analysis* in [12] is not necessarily safe. These issues are discussed in [14, 16] where the impact of transmission by polling (as opposed to interrupt driven) is also outlined. Using FIFO or any work-conserving queuing for messages inside the CAN driver/middleware layers (violating **(a)** in the previous list) is discussed and analyzed in [11, 13, 57]. When the copy time from the message queue to the TxObject cannot be neglected (disobeying **(b)** in the list), the introduced priority inversion is analyzed in [40]. Di Natale and Zeng [15] provides theory for the analysis of systems in which message output at the CAN driver is performed by polling (another break of the rule **(b)**). For the violation to **(c)** in the list, additional delay can be caused by limited number of TxObjects at the CAN controller that are non-abortable. Natale [41] and Khan et al. [61] provide an analysis to these driver configuration issues and controller policies that can lead to (possibly multiple) priority inversions. Di Natale and Zeng [15] provides further insight on the management of TxObjects without preemption and proposes a heuristic for the design of message queuing systems with guaranteed real-time properties. Finally, [58, 59] integrate the effect of these hardware and software limitations with messages that are triggered by both time (periodic) and event (sporadic).

As an example, the analysis for systems with FIFO software queue [13], under the condition that the messages have constrained deadlines ($D \leq T$), is summarized below. The more general case of unconstrained deadlines is discussed in [11]. For a message m_i, the maximum time that it waits in the FIFO queue before it becomes the oldest message (hence ready for priority-based arbitration) is defined as the *buffering delay* of m_i. The buffering delay of m_i is denoted as f_i.

The analysis returns the same worst-case response time bound for all messages $\mathbf{M}$ in the same FIFO queue. It is similar to Eq. (24.6) in that the blocking time is safely bounded to avoid checking multiple instances in the busy period. For each contributing delay to the response time, it makes pessimistic but safe assumptions to derive a correct upper bound. The *first* delay is the blocking time, upper bounded by the maximum between direct blocking time from lower-priority messages or the indirect blocking bounded by the push through interference $C_{\mathbf{M}}^{\max} = \max_{j \in \mathbf{M}} C_j$. The *second* is the interferences from messages from the same FIFO queue. Since messages have a deadline no larger than the period, there can be at most one instance from each message waiting in the queue for a schedulable system. Hence, the maximum interference caused by these messages is upper bounded by $C_{\mathbf{M}}^{\text{SUM}} - C_{\mathbf{M}}^{\min}$ where $C_{\mathbf{M}}^{\text{SUM}} = \sum_{j \in \mathbf{M}} C_j$ and $C_{\mathbf{M}}^{\min} = \min_{j \in \mathbf{M}} C_j$. In this way, among the total transmission time $C_{\mathbf{M}}^{\text{SUM}}$, the maximum amount is also exposed to interference from messages in other queues when m_i has a transmission time $C_i = C_{\mathbf{M}}^{\min}$. The *third* delay is the interferences from messages in other queues. This delay is maximized when we consider the lowest-priority message $m_{L_i} \in \mathbf{M}$ queued in the same FIFO as m_i.

Summating all the above, the queuing delay w_i can be derived from a sufficient condition similar to that in Eq. (24.6) for priority-queued messages:

$$w_i = \max(B_{L_i}, C_{\mathbf{M}}^{\max}) + (C_{\mathbf{M}}^{\mathrm{SUM}} - C_{\mathbf{M}}^{\min}) + \sum_{k \in hp(L_i) \wedge k \notin \mathbf{M}} \left\lceil \frac{w_i + J_k + f_k + \tau_{bit}}{T_k} \right\rceil C_k \tag{24.7}$$

and the response time of m_i is bounded by adding the queuing delay and the transmission time together:

$$R_i = w_i + C_{\mathbf{M}}^{\min} \tag{24.8}$$

In Eq. (24.7), besides the queuing jitter J_k, the buffering delay f_k should be treated as an additional jitter. This is because f_k quantifies the variation from the readiness of m_k to the time it enters the priority-based arbitration (and becomes capable of interfering m_{L_i}). The buffering time f_i of m_i can be bounded as

$$f_i = R_i - C_{\mathbf{M}}^{\min} \tag{24.9}$$

24.3 A Heterogeneous Communication Protocol: FlexRay

24.3.1 Introduction

The FlexRay standard was developed by a consortium including the major car manufacturers and their suppliers, with a stated objective *to support cost-effective deployment of distributed by-wire controls*. It is now defined in a set of ISO standards, ISO 17458-1 to 17458-5 [37]. In addition to the stringent requirements on determinism and short latencies as those for the x-by-wire functions, the definition of FlexRay also was motivated by the large volumes of data traffic from active safety functions.

The upper bound of communication speed in FlexRay is defined as 10 Megabits per second (Mbps), as opposed to 1 Mbps for CAN. Time is divided into communication cycles that are of equal length. Each communication cycle contains up to four segments: *static*, *dynamic*, *symbol window* (to transmit FlexRay-defined symbols for, e.g., maintenance and cold-start cycles), and *network idle time (NIT)* (for clock correction due to, e.g., clock drift), as shown in Fig. 24.3. Clock synchronization is embedded in the standard, using part of the NIT segment.

The *static* segment of the communication cycle enables the transmission of time-critical messages according to a periodic pattern, i.e., with *time-triggered* (TDMA) communication. It is divided into a set of equal-sized time *slots*. The transmission of frames in the static segment is fixed in a given slot, at a given time window. The *dynamic* segment allows for flexible communications. The transmission of frames in the dynamic segment is *event triggered*, arbitrated by their identifiers, where the lowest identifier frames are transmitted first. Frames from different nodes

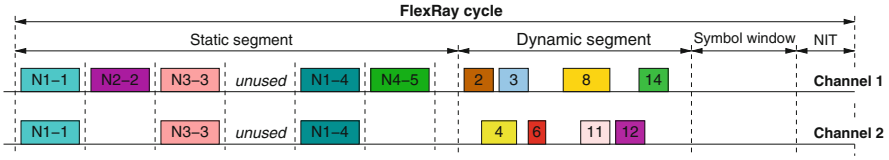


Fig. 24.3 The four segments in a FlexRay communication cycle

can share the same identifier, but they differ in the allowed communication cycle. This flexibility, called *slot multiplexing*, is supported in the most recent FlexRay standard [37]. For increased reliability and timing protection, FlexRay includes specifications of dual channel as well as bus guardians at the node and star level. In a dual-channel configuration, frames for safety-critical communications can be replicated in both channels (as those from node N1 in Fig. 24.3), or the slots can be assigned independently.

The FlexRay bus configuration includes the selection of several parameters, including the length of the communication cycle, the number and length of time slots in the static segment, and the slot time of the dynamic segment. There are several issues that require careful consideration in the definition of the bus configuration: future extensibility, the composability of subsystems, and the possible standardization for reusing of ECU components. Due to the nature of the automotive supply chain and the desire to reuse ECUs on different car platforms, there is a trend of global standardization of these FlexRay bus configuration parameters.

24.3.2 Static Segment

In FlexRay static segment, each node keeps the specification of the time slots for its outgoing and incoming communications in its *local scheduling table*. The local scheduling tables of all nodes shall be consistent (i.e., no two nodes are scheduled to send frames in the same slot of the same communication cycle). In this way, the schedule is *composable* (in the sense that no timing conflicts or interferences arise), each node executes with respect to its own (synchronized) clock, and there is no need for storing a global scheduling table.

Slots that are left free in the (virtual) global table resulting from the composition of the local tables can be used for future extensions. Bus guardians monitor and prevent a node from transmitting outside the allocated time window. This guarantees time protection and isolation from timing faults.

The scheduling of FlexRay systems consists of the scheduling of the task and signal instances in an application cycle. Broadly speaking, there are two possible synchronization patterns between tasks and signals:

- **Asynchronous scheduling.** Such a scheduling model does not require that the job (an instance of the task that produces the signal) and signal schedulers are synchronized. Jobs post data values for the output signals in shared variables.

The communication drivers have the responsibility to pack the signals into frames and fill the registers for the outgoing communication slot. At the receiving side, the received frames are de-packed and written into input registers such that they can be read asynchronously by the reader tasks.

- **Synchronous scheduling.** Different from the asynchronous scheduling model, job executions and frame transmissions are synchronized such that a job must complete before the beginning of the slot that transmits its output signal (with a margin determined by the necessary copy time). It is then necessary to know what job produces the data that is transmitted by a frame and what is the job that reads the data delivered by the frame. It leverages the full benefits of the FlexRay deterministic communication: Scheduling can be arranged to achieve small sampling delays, providing very tight end-to-end latencies and small jitter. Also, equally important, this scheduling model allows to guarantee time determinism and the preservation of the stream of data exchanged over the bus [27].

For synchronous scheduling of signals and tasks, besides packing the signals into frames, designers will need to schedule the software tasks and frames, such that timing constraints including end-to-end deadlines are satisfied. The precedence constraints induced by information passing between tasks and signals and the end-to-end delays associated with control path should be taken into consideration. ILP-based approaches, holistic or two-stage, are studied by Zeng et al. [80], where the tasks are scheduled at job level (each job in the hyperperiod can be scheduled independently). Lukasiewycz et al. [49] provide a framework for scheduling buses and ECUs at the task level, to enable an AUTOSAR compliant system. Besides the timing-related metrics (bandwidth, end-to-end latency), synchronous scheduling is also addressed under other contexts, such as application-level acknowledgment and retransmission scheme [43], robustness to uncertainties in design parameters [24], and message authentication/verification for security enhancement [28]. Also, Han et al. [27] discuss that system-level time-triggered schedules allow the semantics-preserving implementation of distributed control models with a synchronous reactive semantics and develop algorithms for minimizing functional delays to improve control quality.

When considering the asynchronous scheduling, the FlexRay scheduling problem consists in the optimization of the communication scheduling for a set of periodic signal streams, generated at the ECU interface and considered independently from their sender and receiver tasks. For several car manufacturers this is a problem of high practical interest, because the first step in the move to FlexRay is likely to be the remapping of existing CAN communication flows, which are today typically asynchronous with respect to computations. This problem is addressed in a number of papers as follows:

Ding et al. adopt genetic algorithm and its combination with bin packing [22,23]. Schmidt et al. present a two-stage Integer Linear Program (ILP) approach [67], where the first stage packs signals to frames and the second stage determines the schedule from the set of frames. Grenier et al. [26] design a simple heuristic

assuming one signal per frame. Lukasiewycz et al. [48] develop a bin-packing-based approach for allocating signals to slots. Tanasa et al. [69] propose to use message retransmissions on the FlexRay static segment to provide guarantees on reliability. These works focus on the earlier version of FlexRay 2.1 where slot multiplexing is disallowed, hence preventing the share of the same slot across different ECUs. In compliance with the latest FlexRay 3.0 standard, Schenkelaars et al. [66] consider the mapping of frames to slots but assume the signals are already packed to frames. Kang et al. [39] consider the problem of packing signals to frames. Hu et al. [30] adopt a list-scheduling-based approach. Darbandi et al. [9] transform the problem to a strip packing problem. They also propose an ILP-based approach for a direct packing of signals to slots [10].

In the following, the ILP-based approach for synthesizing asynchronous schedule is described. The approach extends the formulation from [48] by considering the slot multiplexing allowed in FlexRay 3.0.

24.3.2.1 ILP-Based Approach for Asynchronous Scheduling

For asynchronous scheduling, the problem the designers are facing is to pack the signals into frames and assign frames to slots such that there is one instance of the signal transmitted within its period, and the number of used slots (the used bandwidth) is possibly minimized.

For the purpose of scheduling in the static segment, it is sufficient to consider the following set of design parameters in the FlexRay bus configuration: (cc, ns, ls), where cc is the number of communication cycles, ns is the number of slots in the static segment of the communication cycle, and ls is the length of the slot in bytes. The application contains a set of ECUs $\mathbb{E}$, each ECU e sending a set of signals $\mathbb{M}_e$. The set of all signals is denoted as $\mathbb{M}$. For each signal $m \in \mathbb{M}$, it is configured with a tuple of parameters (l_m, r_m), where l_m is the length of m in bytes and r_m is the cycle repetition of m. The cycle repetition r_m is essentially the period T_m in the unit of communication cycle: $r_m = T_m / l_{\text{comm}}$, where l_{comm} is the length of the communication cycle in time. r_m shall always be an integer divisor of the number of FlexRay communication cycles cc.

Another option is to evaluate several possible FlexRay configurations in an initial branching of the search procedure. If the number of possible configurations is not very large, it should be possible to explore them and run the optimization framework as an inner loop, comparing the results at the end and choosing the best one with respect to the objective function.

The mapping of signals to slots is encoded in a set of binary variables. For signal m, the base cycle b_m is smaller than r_m ($b_m \in \{0 \ldots r_m - 1\}$); hence the signal to slot mapping is defined as

$$\begin{aligned} &\forall c = 0, \ldots, r_m - 1, \forall s = 0, \ldots, ns - 1, \\ &B_{m,c,s} = \begin{cases} 1, \text{ if } m \text{ is mapped to base cycle } c, \text{ slot } s \\ 0, \text{ otherwise} \end{cases} \end{aligned} \tag{24.10}$$

Another set of binary variables encodes the status of each slot:

$$U_s = \begin{cases} 1, \text{ if slot } s \text{ is used} \\ 0, \text{ otherwise} \end{cases} \tag{24.11}$$

A third set of binary variables encodes the ownership of a slot in a communication cycle:

$$O_{e,c,s} = \begin{cases} 1, \text{ if slot } s \text{ in cycle } c \text{ is owned by ECU } e \\ 0, \text{ otherwise} \end{cases} \tag{24.12}$$

The problem now can be formulated as follows. For the set of constraints, each signal is mapped to one and only one slot in its cycle repetition:

$$\forall m \in \mathbb{M}, \quad \sum_{c=0}^{r_m-1} \sum_{s=0}^{ns-1} B_{m,c,s} = 1 \tag{24.13}$$

The sum of the payloads over all mapped signals within each slot will be upper bounded by the slot size:

$$\forall c = 0, \ldots, cc-1, s = 0, \ldots, ns-1, \quad \sum_{m,\ b \equiv c \pmod{r_m}} l_m \times B_{m,b,s} \leq ls \tag{24.14}$$

If signal m is mapped to communication cycle c and slot s, then m's source ECU e must own that slot:

$$\forall m \in \mathbb{M}_e, c = 0, \ldots, cc-1, b \equiv c \pmod{r_m}, s = 0, \ldots, ns-1, \quad B_{m,b,s} \leq O_{e,c,s} \tag{24.15}$$

If no signal is mapped to slot s in any communication cycle, then the slot ownership is set to null:

$$\forall e \in \mathbb{E}, c = 0, \ldots, cc-1, s = 0, \ldots, ns-1, \quad O_{e,c,s} \leq \sum_{m \in \mathbb{M}_e,\ b \equiv c \pmod{r_m}} B_{m,b,s} \tag{24.16}$$

Each slot in each cycle can only be owned by one ECU:

$$\forall c = 0, \ldots, cc-1, s = 0, \ldots, ns-1, \quad \sum_{e \in \mathbb{E}} O_{e,c,s} \leq 1 \tag{24.17}$$

The slot is used if any of the ECUs owns it:

$$\forall s = 0, \ldots, ns-1, \quad \sum_{e \in \mathbb{E}} \sum_{c=0}^{cc-1} O_{e,c,s} \leq U_s \tag{24.18}$$

The objective is to minimize the number of used slots:

$$\min \sum_{s} U_s \tag{24.19}$$

24.3.3 Dynamic Segment

The FlexRay dynamic segment is partitioned into a number of minislots (MS) that are of equal length. Each frame is assigned a frame identifier (FrameID, or simply ID) within which it can transmit. The dynamic segment is arbitrated in the following way. At the beginning of the dynamic segment, the minislot index is set to one. If there is a ready frame with ID that matches the current slot index, the frame is transmitted. Correspondingly, the dynamic slot is extended to a length equal to the number of minislots needed to transmit the frame, plus one minislot for idle phase (used to separate frame transmissions). Otherwise, the minislot elapses without frame transmission, and the dynamic slot index is incremented before the next minislot starts.

To make sure there is enough time to transmit a frame before the end of the dynamic segment, a parameter $pLatestTx$ is specified for each ECU as the number of minislots in the dynamic segment minus the largest frame size (in minislots). If the current minislot index is larger than $pLatestTx$ (LTx for short in the following), then no frame can start transmission, and all the ready frames are delayed to the next communication cycle.

In the example of Fig. 24.4, five dynamic frames are transmitted over a FlexRay channel. Two frames, m_2 and m_4, share the same ID. At the beginning of the dynamic segment, the dynamic slot index is initialized as 1, and the controller checks whether there is a frame with FrameID 1 ready to be transmitted. m_1, and, consequently, m_2 and m_3, is transmitted in the first communication cycle. However, m_5 with ID 5 cannot be transmitted since there is not enough room to accommodate it and its transmission is delayed to the next cycle (assuming it is allowed to transmit there). In the second communication cycle, there is no frame with ID 1; thus the first dynamic slot is collapsed to one minislot and m_4, which cannot be transmitted

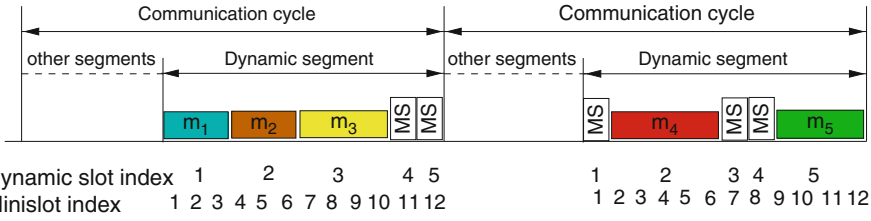


Fig. 24.4 An example of FlexRay dynamic frame transmission

in the first communication cycle because of the transmission of another frame m_2 with the same ID 2, is transmitted, followed by a minislot for indexes 3 and 4 and the transmission of m_5 with ID 5.

For the purpose of scheduling and analysis in the dynamic segment, the FlexRay bus configuration is captured with a list of parameters $(l_{\text{comm}}, l_{\text{ST}}, n_{\text{MS}}, l_{\text{MS}})$, where l_{comm} is the length of the FlexRay communication cycle, l_{ST} is the length of the static segment, n_{MS} is the number of minislots in the dynamic segment, and l_{MS} is the length of the minislot in time (the length of the dynamic segment is $l_{\text{DYN}} = n_{\text{MS}} \cdot l_{\text{MS}}$). In the design flows in the automotive industry, these parameters (in particular the communication cycle and the slot size) are often defined and standardized based on the need to reuse legacy components and standardize configurations, which is likely to induce carmakers to freeze the definition of these parameters for their product lines.

A set of $CC^{\max}$ (of value that is a power of 2, i.e., 1, 2, 4, 8, 16, 32, 64) communication cycles constitute a hyperperiod, for which the scheduling table is specified and repeated. Each dynamic frame $m_i \in \mathbb{M}$ is characterized by a tuple of parameters $\{N_i, T_i, J_i, D_i, C_i\}$, where N_i is its source ECU, T_i is its period, J_i denotes its release jitter (the maximum delay with respect to the periodic activation event), D_i is the deadline, and C_i is the transmission time (the time m_i occupies when transmitted on the bus, including the one minislot needed for the idle phase). For convenience, C_i is defined in terms of the (integer) number of l_{MS}. For example, $C_i = 5 \cdot l_{\text{MS}}$ for frame m_4 in Fig. 24.4. The deadline D_i is assumed to be arbitrary, i.e., it can be smaller, equal, or lager, compared to T_i. The worst-case response time R_i of m_i is the maximum difference between the finish time and the arrival time. For each m_i, LTx_i indicates the latest minislot index in which it is allowed to start transmission. The set of frames with lower ID than m_i is denoted as $lf(m_i) = \{m_j | ID_j < ID_i\}$. Also, the set of frames with lower ID than m_i plus itself is denoted as $le(m_i) = lf(m_i) \bigcup \{i\}$.

In the latest FlexRay 3.0 standard [37], *slot multiplexing* is added as a new feature, which allows different frames sent in different cycles to share the same ID. With slot multiplexing, each frame m_i is assigned with two attributes, base cycle b_i and cycle repetition r_i. Then m_i can only transmit in every r_i-th communication cycle starting at the b_i-th cycle. r_i only assumes a value as a power of 2 that is no greater than $CC^{\max}$, and $b_i \in \{0 \ldots r_i - 1\}$. As a special case, if m_i exclusively occupies the slot (i.e., m_i is the only message allowed to transmit in its slot for all communication cycles), like those without slot multiplexing, and then $b_i = 0$ and $r_i = 1$. Obviously, to avoid infinite waiting for m_i, it shall be $T_i \geq r_i \times l_{\text{comm}}$.

Since the scheduling in FlexRay dynamic segment is not work conserving, i.e., the bus may be left idle even if a message is ready, its timing analysis is inherently more difficult compared to other fixed-priority-based protocols like CAN. *First*, even if there is no message to be sent in a particular slot, it will occupy one minislot. *Second*, if more than one instance of a message is ready, only one of them can be sent in one dynamic segment. *Third*, a message may only be ready after its slot has

started. *Fourth*, there are not enough minislots to transmit a message as its LTx has elapsed. *Lastly*, due to slot multiplexing, the pending messages may not be transmitted in the current communication cycle.

In the following, the timing analysis is discussed in two cases: with or without slot multiplexing. The case of no slot multiplexing is first discussed because of its relative simplicity.

24.3.3.1 Timing Analysis Without Slot Multiplexing

When the feature of slot multiplexing is not used, i.e., the parameters $b_i = 0$ and $r_i = 1$ for all frames m_i, it essentially means that a frame can be transmitted in any cycle.

FlexRay dynamic frames are transmitted in the order of the IDs (priorities) of ready frames and without preemption: If a frame becomes ready while another frame with higher ID is being transmitted (after its slot passed), it cannot preempt the lower-priority frame and must wait until the next cycle. Like other systems with non-preemptive fixed-priority scheduling (such as CAN), the response time analysis is based on the calculation of the *busy period* of level-i, denoted as t_i. The busy period is the worst-case time interval that starts from the critical instant for an instance of m_i queued at $t = 0$ with jitter J_i, where the bus is always busy transmitting frames with priority higher than or equal to m_i except for a possible initial blocking B_i.

Here, B_i denotes the longest blocking delay that happens in the communication cycle when m_i becomes ready. In the worst case, the slot with index i starts as soon as possible in the cycle where no frame with ID lower than i is sent, and m_i is queued right after that. Thus, the worst-case blocking B_i is

$$B_i = l_{\text{comm}} - (l_{\text{ST}} + (ID_i - 1)l_{\text{MS}}) \tag{24.20}$$

Also, for $m_j \in lf(m_i)$, $n_j^{(k)}$ denotes the maximum number of instances of m_j activated in the busy period for the k-th iteration $t_i^{(k)}$. $n_i^{(k)}$ is the number of instances of m_i activated in the busy period (before the one considered for the analysis). The vector $\mathbf{n}^{(k)}$ is defined as $\mathbf{n}^{(k)} = \{n_j^{(k)}\}, j \in le(m_i)$. The function $f_i(\mathbf{n}^{(k)})$ gives the worst-case interference caused by the static segment and the transmission of dynamic frames $m_j \in le(m_i)$ (each activated $n_j^{(k)}$ times). The computation for $f_i(\mathbf{n}^{(k)})$ is discussed in the later part of the section.

The overall length of the busy period can be found as the fixed-point solution of the iterative procedure:

$$\begin{cases} t_i^{(0)} = B_i \\ t_i^{(k+1)} = B_i + C_i + f_i(\mathbf{n}^{(k)}) \end{cases} \tag{24.21}$$

For the iterations in the computation of the busy period t_i, $\mathbf{n}^{(k)}$ is

$$\begin{cases} n_j^{(k)} = \left\lceil \dfrac{J_j + t_i^{(k)}}{T_j} \right\rceil, \forall j \in lf(m_i) \\ n_i^{(k)} = \left\lceil \dfrac{J_i + t_i^{(k)}}{T_i} \right\rceil - 1 \end{cases} \tag{24.22}$$

Inside the busy period t_i, up to $q_i^{\max}$ instances of m_i are ready for transmission:

$$q_i^{\max} = \left\lceil \frac{J_i + t_i}{T_i} \right\rceil \tag{24.23}$$

The worst-case response time of m_i is the maximum among those $q_i^{\max}$ instances in the busy period. These instances are indexed as $q = 1, \ldots, q_i^{\max}$. The longest time from the start of the busy period to the time the q-th instance starts transmission is calculated by the following iterative formula:

$$\begin{cases} w_i^{(0)}(q) = B_i \\ w_i^{(k+1)}(q) = B_i + f_i(\mathbf{n}^{(k)}) \end{cases} \tag{24.24}$$

where $\mathbf{n}^{(k)}$ is given as

$$\begin{cases} n_j^{(k)} = \left\lceil \dfrac{w_i^{(k)}(q) + J_j}{T_j} \right\rceil, \forall j \in lf(m_i) \\ n_i^{(k)} = q - 1 \end{cases} \tag{24.25}$$

The response time of the q-th instance is

$$R_i(q) = J_i + w_i(q) - (q-1)T_i + (C_i - l_{\mathrm{MS}}) \tag{24.26}$$

where l_{MS} is subtracted from C_i as a frame is considered received before the idle phase (whose length is one minislot).

Go back to the computation of $f_i(\mathbf{n}^{(k)})$, the function computing the worst-case delay caused by frames in $le(m_i)$ with known vector $\mathbf{n}^{(k)}$ of activated instances. Since each previous instance of m_i delays the transmission of the following instance of m_i by at least one cycle, their occurrences will produce a delay equal to at least $n_i^{(k)}$ communication cycles. Consider the frames in $lf(m_i)$. Let $s_{\mathrm{cycle}}^{(k)}$ be the number of communication cycles the $(n_i^{(k)} + 1)$-th instance of m_i has to wait because of interference from frames in $lf(m_i)$ and $r_{\mathrm{cycle}}^{(k)}$ be the time from the start of the last cycle to the beginning of the transmission of the $(n_i^{(k)}+1)$-th instance of m_i. $f_i(\mathbf{n}^{(k)})$ can be expressed as

$$f_i(\mathbf{n}^{(k)}) = (n_i^{(k)} + s_{\text{cycle}}^{(k)}) l_{\text{comm}} + r_{\text{cycle}}^{(k)} \tag{24.27}$$

The computation of s_{cycle} and r_{cycle} is demonstrated to be NP hard [65, 68]. *For simplicity, from now on the iteration index k is dropped from* $\mathbf{n}$, s_{cycle} *and* r_{cycle}.

Next, an overview is given on the calculation of the exact value of $f_i(\mathbf{n})$. The calculation is based on the solution of two ILPs [65], which is generally time-consuming especially for large problem sizes. Hence, a load-based heuristic and a heuristic leveraging the results on bin-covering problem are summarized afterward.

The calculation of $f_i(\mathbf{n})$ can be viewed as a constrained version of the *bin-covering problem*. A bin-covering problem is to maximize the number of bins using a given list of items with known weights, such that each bin is at least filled to a minimum capacity (the sum of the weights from items packed to the bin is no smaller than the minimum bin capacity). To calculate $f_i(\mathbf{n})$, after the number of instances n_j from each frame $m_j \in lf(m_i)$ is calculated, these instances are used to cover bins (communication cycles) with minimum capacity $LTx_i \cdot l_{\text{MS}}$. Each frame m_j has n_j instances, and its weight is C_j. The problem also contains additional constraints such as (24.34) and (24.35) below.

Exact Solution for $f_i(\mathbf{n})$

Below a brief summary is given on the ILP optimization formulation proposed in [65] to find the exact solutions. The number of busy communication cycles s_{cycle} can be bounded by the total number of frame instances in $lf(m_i)$:

$$s_{\text{cycle}} \le s_{\text{cycle}}^{\text{ub}} = \sum_{j \in lf(m_i)} n_j \tag{24.28}$$

or better, by using the heuristic in (24.41) presented in the following subsections. The number of binary variables in the ILP formulation depends linearly on the maximum number of busy communication cycles; thus using a tighter bound like (24.41) can greatly reduce the complexity.

A set of binary variables defines the transmission of the instances of frames in $lf(m_i)$ in the cycles:

$$\forall j \in lf(m_i), n \le n_j, s \le s_{\text{cycle}}^{\text{ub}}, \quad x_{j,n,s} = \begin{cases} 1 & \text{if the } n\text{-th instance of } m_j \\ & \text{is sent in cycle } s \\ 0 & \text{otherwise} \end{cases} \tag{24.29}$$

Another set of binary variables denotes whether cycle s is busy transmitting frames $\in lf(m_i)$ or not:

$$\forall s \le s_{\text{cycle}}^{\text{ub}}, \quad y_s = \begin{cases} 1 & \text{if the } s\text{-th cycle is busy} \\ 0 & \text{otherwise} \end{cases} \tag{24.30}$$

The total load $N_{i,s}$ from $lf(m_i)$ and idle minislots with slot index $< ID_i$ in cycle s is

$$N_{i,s} = \sum_{j \in lf(m_i)} \sum_{n \le n_j} x_{j,n,s} \cdot C_j + \sum_{j < ID_i} (1 - \sum_{n \le n_j} x_{j,n,s}) l_{\mathrm{MS}} \tag{24.31}$$

A set of constraints encodes the FlexRay protocol requirements. First, in any busy cycle s, $N_{i,s}$ must be no smaller than the length of LTx_i minislots:

$$\forall s \le s^{\mathrm{ub}}_{\mathrm{cycle}}, \quad N_{i,s} \ge LTx_i \cdot l_{\mathrm{MS}} \cdot y_s \tag{24.32}$$

Second, each frame instance is transmitted at most once:

$$\forall j \in lf(m_i), n \le n_j, \quad \sum_{s \le s^{\mathrm{ub}}_{\mathrm{cycle}}} x_{j,n,s} \le 1 \tag{24.33}$$

and each frame ID is transmitted at most once in each cycle

$$\forall j \in lf(m_i), s \le s^{\mathrm{ub}}_{\mathrm{cycle}}, \quad \sum_{n \le n_j} x_{j,n,s} \le 1 \tag{24.34}$$

Third, each instance of m_j is sent no later than the minislot of index LTx_j, formulated using the "big-M" formulation:

$$\forall j \in lf(m_i), n \le n_j, s \le s^{\mathrm{ub}}_{\mathrm{cycle}}, \; N_{j,s} < LTx_j \cdot l_{\mathrm{MS}} \cdot y_s + M(1 - x_{j,n,s}) \tag{24.35}$$

Here M is a large-enough constant, e.g., l_{DYN}.

To find the maximum number of busy communication cycles s_{cycle}, an optimization problem is solved, with objective function in (24.36) and constraints in (24.32), (24.33), (24.34), and (24.35):

$$s_{\mathrm{cycle}} = \max(\sum_{s \le s^{\mathrm{ub}}_{\mathrm{cycle}}} y_s) \tag{24.36}$$

With the solution to s_{cycle}, the value of r_{cycle} can be determined by maximizing the communication load in the $(s_{\mathrm{cycle}} + 1)$-th cycle after filling up the first s_{cycle} cycles:

$$r_{\mathrm{cycle}} = \max(l_{\mathrm{ST}} + N_{i,s_{\mathrm{cycle}}+1}) \tag{24.37}$$

Load-based Heuristic Solution for $f_i(\mathbf{n})$.

The basic idea to approximate $f_i(\mathbf{n})$ is to assume that the s_{cycle} communication cycles are always filled with the minimum amount of load from frames in $lf(m_i)$.

Hence, the concept of the *minimum serviced load* is defined as the minimum transmission time that is needed in addition to the idle minislots to fill a communication cycle bin. It can be calculated as

$$\begin{aligned} p_i &= LTx_i \cdot l_{\mathrm{MS}} - (ID_i - 1)l_{\mathrm{MS}} \\ &= (LTx_i - ID_i + 1)l_{\mathrm{MS}} \end{aligned} \tag{24.38}$$

This may also be derived by manipulating constraint (24.32):

$$\begin{aligned} N_{i,s} &= \sum_{j \in lf(m_i)} \sum_{n \le n_j} x_{j,n,s} C_j + \sum_{j < ID_i, j \notin lf(m_i)} l_{\mathrm{MS}} + \sum_{j \in lf(m_i)} (1 - \sum_{n \le n_j} x_{j,n,s}) l_{\mathrm{MS}} \\ &= \sum_{j \in lf(m_i)} \sum_{n \le n_j} x_{j,n,s}(C_j - l_{\mathrm{MS}}) + (ID_i - 1)l_{\mathrm{MS}} \\ &\ge LTx_i \cdot l_{\mathrm{MS}} \cdot y_s \end{aligned} \tag{24.39}$$

With $C_i' = C_i - l_{\mathrm{MS}}$, constraint (24.32) is equivalent to

$$\sum_{j \in lf(m_i)} \sum_{n \le n_j} x_{j,n,s} \cdot C_j' \ge p_i \cdot y_s \tag{24.40}$$

An upper bound on s_{cycle} can be derived by considering a bin-packing problem formulation, where (24.32) (or equivalently (24.40)) and (24.33) are respected but constraints (24.34) and (24.35) are ignored [65]. With the notation K_i as

$$K_i = \frac{\sum_{j \in lf(m_i)} (n_j \cdot C_j')}{p_i} \tag{24.41}$$

it is

$$s_{\mathrm{cycle}} \le \lfloor K_i \rfloor \tag{24.42}$$

The upper bound on $f_i(\mathbf{n})$ is

$$f_i(\mathbf{n}) \le (n_i + \lfloor K_i \rfloor) l_{\mathrm{comm}} + l_{\mathrm{ST}} + (ID_i - 1)l_{\mathrm{MS}} + (K_i - \lfloor K_i \rfloor) p_i \tag{24.43}$$

However, the upper bound in Eq. (24.43) is in general loose, since constraint (24.34) that requires each frame identifier is used at most once in each cycle is ignored. With this observation, constraint (24.34) is brought back into consideration: at most one instance of any frame $m_j \in lf(m_i)$ can be packed in each communication cycle.

Algorithm 1 reflects this idea and gives a tighter upper bound $f_i^H(\mathbf{n})$ on $f_i(\mathbf{n})$ than Eq. (24.43). It uses a *load variable* L to denote the available requested transmission time within a communication cycle. The loop from Lines 2–15

implements the iterative procedure to calculate L and s. It tries to fill each bin (communication cycle) starting from cycle $s = 0$. In each cycle s, one instance from each $m_j \in lf(m_i)$ that has $n_j > 0$ is added to get the load variable L (Lines 3–8). L is the maximum amount of time that is available for transmission in the communication cycle s. By adding only a term $C'_j = C_j - l_{\text{MS}}$ for each $m_j \in lf(m_i)$, at most one instance from each frame m_j can be transmitted in this cycle. If $L \geq p_i$, the cycle s is filled, the bin capacity p_i is subtracted from the load variable L, and the iteration continues to the next cycle (Lines 9–11). The reason is that in the worst-case scenario, only p_i of these loads is transmitted in the communication cycle and the maximum amount of *remaining loads* is delayed to the next cycles. If $L < p_i$, there is not enough load to fill the bin, and m_i is transmitted in the current communication cycle, and the iteration stops (Lines 12–13). Then, s is assigned to s^H_{cycle}, and r^H_{cycle} is calculated as $l_{\text{ST}} + (ID_i - 1)l_{\text{MS}} + L$, assuming L as the additional load in the communication cycle in which the $(n_i + 1)$-th instance of m_i is transmitted (Line 16). Finally, $f_i(\mathbf{n})$ is calculated as the length of $(n_i + s_{\text{cycle}})$ communication cycles plus r_{cycle} (Line 17).

Algorithm 1 Algorithm to compute the upper bound $f_i^H(\mathbf{n})$ on $f_i(\mathbf{n})$

1: $p_i = (LTx_i - ID_i + 1)l_{\text{MS}}$, $L = 0$, $s = 0$
2: **while** true **do**
3: **for** each frame $j \in lf(m_i)$ **do**
4: **if** $n_j > 0$ **then**
5: $L = L + C_j$
6: $n_j = n_j - 1$
7: **end if**
8: **end for**
9: **if** $L \geq p_i$ **then**
10: $s = s + 1$
11: $L = L - p_i$
12: **else**
13: break
14: **end if**
15: **end while**
16: $s^H_{\text{cycle}} = s$, $r^H_{\text{cycle}} = l_{\text{ST}} + (ID_i - 1)l_{\text{MS}} + L$
17: $f_i^H(\mathbf{n}) = (n_i + s^H_{\text{cycle}})l_{\text{comm}} + r^H_{\text{cycle}}$

The value of $f_i^H(\mathbf{n})$ returned by Algorithm 1 is proven to be no smaller than the exact value $f_i(\mathbf{n})$ [83]. Essentially s^H_{cycle} is solved with a more relaxed problem than the exact solution s_{cycle}. It ignores the constraints (24.32) and (24.35). The complete proof is documented in [83]. Hence, the algorithm is a pessimistic but safe procedure in that the resulted worst-case response time R_i is always an upper bound for the actual response time.

Bin-Covering-Based Heuristic Solution for f_i(n).
Alternatively, Tanasa et al. [68] apply recent theoretical advances [38] to approximate the upper bounds on the optimal solution for the bin-covering problem. It takes an input parameter ϵ to define the precision of the results. The details of how the bin-covering heuristic is solved can be found in [68]. It is reported that [68] the approach provides improvement over the load-based heuristic when $\epsilon \leq 1/16$ and is comparable when $\epsilon = 1/8$. Hence, a value between $1/16$ and $1/8$ for ϵ can provide the right balance between efficiency and quality.

24.3.3.2 Extension to Slot Multiplexing

The generalization of the two techniques to slot multiplexing is now discussed. For this purpose, p_i is extended with an additional parameter cc, i.e., $p_i(cc)$, to denote the minimum serviced load for communication cycle cc. Similarly, $lf(m_i, cc)$ denotes the set of messages in $lf(m_i)$ which can be transmitted in cc, i.e., $lf(m_i, cc) = \{m_j | m_j \in lf(m_i), cc \equiv b_j \pmod{r_j}\}$.

Consider an example as presented in [68], which contains five frames with their cycle repetition and base cycle in Table 24.2. Figure 24.5 illustrates the communication cycles that each frame is allowed to transmit.

Load-Based Heuristic
Slot multiplexing provides flexibility and efficiency for scheduling but also introduces new challenges to the timing analysis. Each communication cycle should be regarded differently, due to the facts that the message m_i under analysis and those with lower ID (in $lf(m_i)$) cannot be transmitted in every cycle.

Table 24.2 Frame parameters of an example

	m_1	m_2	m_3	m_4	m_5
Cycle repetition	1	2	4	8	2
Base cycle	0	0	0	0	0

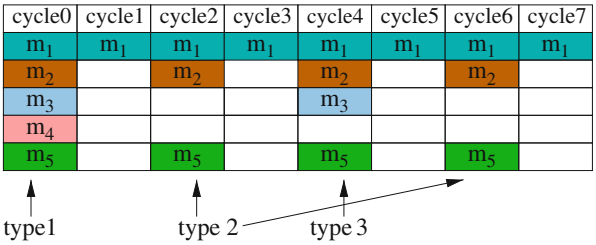


Fig. 24.5 The allowed communication cycles for the frames in Table 24.2

- First, the first complete communication cycle cc_0 after the start of the busy window can be any one in the hyperperiod. Hence, in the analysis, it is sufficient to enumerate the possible values $\{0...CC^{\max}-1\}$ for cc_0.
- Second, it is insufficient to use a single load variable since the load accumulated in one cycle may not be delayed to the next cycles. This can be illustrated with a simple example that m_i can be transmitted in every cycle, and each message m_j in $lf(m_i)$ has $r_j = 2$ and $b_j = 0$. Even if the accumulated load from $lf(m_i)$ in cycle 0 can be larger than $2p_i$, it is not deferrable to cycle 1, and m_i can be sent in cycle 1. The solution to this issue is explained later.
- Third, the minimum serviced load p_i in each cycle is different as m_i may not be transmitted in certain cycles. A conservative estimate is that for those communication cycle $cc \mod r_i \neq b_i$, $p_i(cc)$ is set to be 0. This can be improved to $p_i(cc) = \min_{j \in lf(m_i,cc)} p_j(cc)$, since any of the messages in $lf(m_i)$ could still start transmission if less load was serviced [63].

To analyze the effects of slot multiplexing on the interference that messages in $lf(m_i)$ may introduce (the second challenge above), a set of load variables $L_{b,r}$ is added to denote the cycle-dependent load, where b is the base cycle and r is the repetition factor. At a given cycle cc during the analysis, the transmission time C_j of m_j is added to the load variable L_{b_j,r_j} only if $cc \equiv b_j \pmod{r_j}$, to reflect the fact that m_j is only allowed to transmit in such communication cycles. Hence, the total available load $L(cc)$ at cc is

$$L(cc) = \sum_{b,r:\ cc \equiv b \pmod{r}} L_{b,r} \tag{24.44}$$

If $L(cc) < p_i(cc)$, then there is not enough load to further delay m_i, and m_i will be sent in the earliest cycle $\geq cc$ that it is permitted to transmit. Otherwise, m_i will be delayed, and the minimum serviced load should be subtracted.

Since $L(cc)$ is in general composed of loads from several suitable $L_{b,r}$ variables, there is an additional question about which $L_{b,r}$ should $p_i(cc)$ be subtracted from. Neukirchner et al. [63] observe that the repetition factors are only allowed to be a power of 2, which helps to simplify the problem. Because of this restriction in the FlexRay specification, any load variable coincides with several load variables of a lower repetition factor. For example, $L_{3,4}$ always coincides with $L_{1,2}$ and $L_{0,1}$. To maximize the $L(cc)$ for every cc, it is sufficient to maximize the $L_{b,r}$ variable with the smallest cycle repetition r. This can be achieved by subtracting the minimal serviced load $p_i(cc)$ first from the available $L_{b,r}$ with the highest cycle repetition.

Bin-Covering-Based Heuristic

The heuristic based on bin-covering approximation algorithm can also be extended to slot multiplexing. However, the problem is no longer a traditional bin-covering problem. Rather, the problem becomes what was called bin-covering problem with conflicts [68], as not all messages (items) can be transmitted in every

communication cycle (bin). The number of types of bins P is determined by the distinct communication cycles in a hyperperiod, where two communication cycles are considered distinct if the set of messages they can carry are different. With this understanding, the problem then can be reformulated as bin-covering problem with specific requirements on the number of bins to be packed for each of the P types. For example, in Fig. 24.5, there are three types of bins for the purpose of analyzing m_5's response time: type 1 containing cycle 0 where all higher-priority messages m_1–m_4 can transmit, type 2 for cycles 2 and 6, and type 3 for cycle 4.

24.4 Packet-Switched Networks: Ethernet

24.4.1 Introduction

In addition to traditional buses such as CAN or FlexRay, packet-switched Ethernet will be used in next-generation automotive communication architectures. Ethernet's superior bandwidth and flexibility make it ideal to address the high communication demands of, for example, Advanced Driver Assistance Systems (ADASs), infotainment systems, and ECU flashing. As a switched network, Ethernet provides a scalable, high-speed, and cost-effective communication platform, which allows arbitrary topologies.

Ethernet evolved from a shared bus communication medium with Carrier Sense Multiple Access/Collision Detection (CSMA/CD)-based link access scheme to a switched network. Frame collisions in CSMA/CD were resolved by a binary exponential backoff algorithm which picked a random delay until a retransmission could be started after a collision. This deemed CSMA/CD unsuitable for real-time systems with tight latency or jitter requirements. Switched Ethernet made CSMA/CD obsolete. In switched Ethernet, contention is moved into the switches, where a scheduler has full control over each output port. This enables the implementation of elaborate link schedulers, which allow the derivation of real-time guarantees. Today, Ethernet installations (including the automotive domain) are almost always switched. Hence, in the following, we will refer to switched Ethernet as standard Ethernet.

In the automotive context, Ethernet is anticipated to serve as an in-vehicle communication backbone, where it must be able to transport traffic streams of mixed criticality. This requires Quality of Service (QoS) mechanisms, in order to provide deterministic timing guarantees for critical traffic. Standard Ethernet (IEEE 802.1Q) introduced eight traffic classes. These classes can be used to prioritize traffic, which is typically implemented by a Static-Priority Non-Preemptive (SPNP) scheduler at each output port in each switch and end point. This limited number of classes requires that multiple traffic streams share a class, making streams of equal priority indistinguishable to the scheduler. Traffic within a shared class is usually scheduled in First-In First-Out (FIFO) order.

Compared to CAN or FlexRay, Ethernet exhibits complex timing behavior, as each switch output port is a point of arbitration, which adds delay to the overall

end-to-end latency. While mature formal performance analysis techniques have been established for CAN and FlexRay, such techniques are even more required for Ethernet before it can be used in timing- and safety-critical systems. This will become even more important in the context of highly automated and autonomous driving. In this section, we use Compositional Performance Analysis (CPA) (see ▸ Chap. 23, "CPA: Compositional Performance Analysis" and [29]) to derive worst-case performance bounds for Ethernet.

24.4.2 Modeling Ethernet Networks for Performance Analysis

Before timing guarantees can be derived for an Ethernet network, the components of this network must be mapped to the CPA system model (cf. Sect. 2 in ▸ Chap. 23, "CPA: Compositional Performance Analysis"). This mapping process is explained in detail in [19]. Here, a brief summary covering the essential steps, using Figs. 24.6 and 24.7 as illustration, is presented.

Figure 24.6 shows an Ethernet model comprising two switches and four ECUs. A sequence of related Ethernet frames between a source and one (or more) destination(s) is called an Ethernet traffic stream. There are two traffic streams in the network: a unicast stream from ECU0 to ECU3 and a multicast stream from ECU2 to ECU1 and ECU3.

In order to map the Ethernet model to the CPA system model, resources, tasks, and event models must be identified. Resources model points of contention. In Ethernet, contention between individual frames happens at the switches. Inside a switch there are several delay sources. At the input port, there is *input queuing delay*, the switch fabric adds *forwarding delay*, and at the output port, there is *output queuing delay*. Contemporary switches are fast enough that input queuing delay and forwarding delay only have a negligible impact on the overall timing guarantees. Hence, these delays can be ignored or approximated by constant terms. The output queuing delay considers the time it takes to transmit a given frame, including the interference from other frames. Consequently, switch output ports are modeled by CPA resources. The scheduling policy of these resources is determined by the switch port's scheduling mechanism. Additionally, here is *transmission delay* on the link between switches. This delay corresponds to the propagation delay of electric signals on the link's wire and can also be modeled by a constant term.

The transmission of a frame via an output port of a switch is modeled in CPA as the execution of a task on the port's resource. An Ethernet traffic stream is modeled as a chain of dependent frames (tasks) according to its path through the network

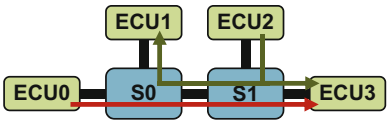


Fig. 24.6 Ethernet model

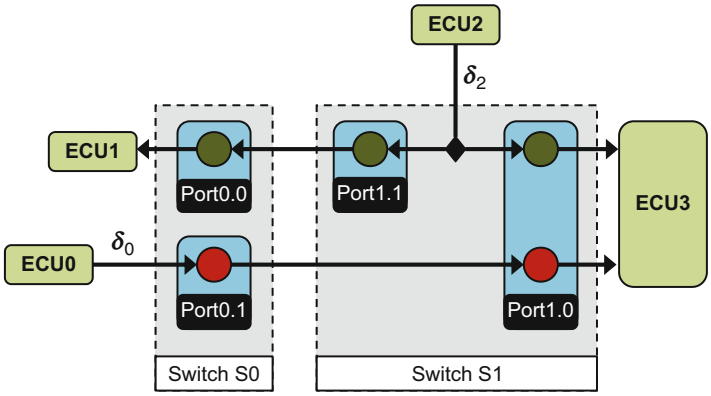


Fig. 24.7 CPA system model for the Ethernet model from Fig. 24.6

(see Fig. 24.7). This chain may fork to model multicast or broadcast trees. On each resource, a task consumes service according to its execution time bounds, which are derived from the best-case and worst-case transmission times of its corresponding Ethernet frame. The transmission time of a frame is defined to be the time it takes the frame to be transmitted without any interference from other frames. For a frame of traffic stream i with maximum/minimum payload $p_i^{-/+}$, the best-case and worst-case transmission times C_i^- and C_i^+ can be computed to

$$C_i^{+/-} = \frac{42\,\text{bytes} + \max\left\{42\,\text{bytes}, p_i^{+/-}\right\}}{r_{TX}} \tag{24.45}$$

where r_{TX} is the transmission rate of the port that transmits the frame. The constant terms correspond to the protocol overhead. The first 42 bytes account for preamble (7 bytes), start of frame delimiter (1 byte), destination and source Media Access Control (MAC) address (both 6 bytes), IEEE 802.1Q tag (4 bytes), EtherType (2 bytes), frame check sequence (4 bytes), and inter-frame gap (12 bytes). The second 42 bytes account for the fact that there is a minimum Ethernet frame size of 84 bytes and that the payload must be padded if necessary.

Frame arrivals (and emissions) are modeled by event models. These models come from either external sources or from dependent frames.

Figure 24.7 shows the corresponding CPA model of the Ethernet model from Fig. 24.6. As can be seen, the output ports of both switches are modeled as resources (light blue boxes with rounded corners). Both traffic streams are modeled by a chain of tasks (red and green circles) reflecting their paths through the network. Notice that the green path originating at ECU2 splits into a multicast tree. ECU0 and ECU2 inject frames into the network according to the event models δ_0 and δ_2 (respectively). This model can then be analyzed with CPA's iterative approach.

24.4.3 Analysis of Standard Ethernet (IEEE 802.1Q)

In order to derive upper bounds on the worst-case performance of Ethernet networks, an analysis which captures all delay effects on the CPA resources that model the switch output ports must be developed. This analysis will then be used in the local analysis step of the CPA loop to derive worst-case frame transmission latencies on each output port.

Definition 1. A frame's transmission latency is the time interval, which starts when the frame has been received at an input port and ends when it has been transmitted entirely from an output port. The transmission latency includes all timing effects from interfering traffic streams.

In the context of the model transformation from Sect. 24.4.2, the transmission latency of a frame corresponds to the response time of a task.

When deriving formal performance guarantees, the worst-case transmission latency of the frames of a given traffic stream i (among all for stream i's possible frames transmission latencies) is of particular interest. For non-preemptive scheduling (such as standard Ethernet), it has been shown that, in order to find this worst-case transmission latency, the transmission latencies of all frames of stream i in its longest scheduling horizon must be evaluated (cf. [12]). The scheduling horizon of a traffic stream i is the time a switch port is busy processing frames of stream i, including interference from frames of other traffic streams (cf. Sect. 2.2.1 in ▸ Chap. 23, "CPA: Compositional Performance Analysis" and [17]). Particularly, the worst-case transmission latency of the q-th frame of traffic stream i can be derived from its worst-case multiple activation queuing delay $Q_i(q, a_i^q)$ (cf. Eq. (9) in ▸ Chap. 23, "CPA: Compositional Performance Analysis").

Definition 2. Assuming that the q-th frame of a traffic stream i arrives at time a_i^q at a switch output port, its worst-case multiple activation queuing delay $Q_i(q, a_i^q)$ is the time interval, which starts with the arrival of the first frame of stream i that initiates the scheduling horizon and ends when the q-th frame can be transmitted (i.e., it does not include the transmission of the q-th frame).

Note that, in contrast to the multiple activation queuing delay $Q_i(q)$ introduced in Sect. 2.2.1 in ▸ Chap. 23, "CPA: Compositional Performance Analysis", the queuing delay in the Ethernet context additionally depends on the arrival time a_i^q of the q-th frame. This is due to the FIFO scheduling of frames with equal priority and will be explained later in this section. The arrival time a_i^q of the q-th frame of stream i is measured relative to the beginning of the scheduling horizon.

As stated in Sect. 24.4.2, it is assumed that the queuing delay of a given frame at a switch output port accounts for all delays induced by interfering traffic streams. The amount of interference from other traffic streams depends on the output port's scheduling policy. In standard Ethernet, traffic streams are categorized into (up to)

eight traffic classes, which correspond to priority levels. Inside each output port there is a set of FIFO queues, one for each traffic class. These FIFO queues are served by an SPNP scheduler. Consequently, to calculate the worst-case queuing delay $Q_i(q, a_i^q)$ in standard Ethernet, all blocking effects, which can occur in this combination of FIFO and SPNP scheduling, must be considered.

Lower-priority blocking: In non-preemptive scheduling, a frame which started transmitting is guaranteed to finish without interruption. Hence, a frame of traffic stream i can experience blocking from at most one lower-priority frame, if this lower-priority frame started transmitting just before the arrival of the first frame of traffic stream i [21]:

$$I_i^{LPB} = \max_{j \in lp(i)} \left\{ C_j^+ \right\} \tag{24.46}$$

where $lp(i)$ is a function yielding the set of all traffic streams whose priority is lower than that of stream i.

Higher-priority blocking: In any time interval of length Δt, a frame of traffic stream i can experience blocking from all frames of higher-priority streams, which arrive during Δt, i.e., before the frame of stream i can be transmitted [21]:

$$I_i^{HPB}(\Delta t) = \sum_{j \in hp(i)} \eta_j^+(\Delta t + \epsilon) C_j^+ \tag{24.47}$$

where $hp(i)$ is a function yielding the set all traffic streams whose priority is higher than that of stream i. Recall from Sect. 2.1.2 in ▸ Chap. 23, "CPA: Compositional Performance Analysis" that $\eta^+(\Delta t)$ yields an upper bound on the number of events, i.e., frame arrivals, in any half open time interval of length Δt. As the multiple activation queuing delay $Q_i(q, a_i^q)$ covers the time *until* the q-th frame can be transmitted, higher-priority frames arriving exactly at the end for Δt can also interfere with the q-th frame. We model this by adding an infinitesimal small time ϵ to Δt to cover the *closed* time interval $[t, t + \Delta t]$. In practice, ϵ corresponds to a bit time.

Same-priority blocking: As frames of identical priority are processed in FIFO order, frames of traffic stream i can experience blocking from frames of other traffic streams with the same priority as stream i. Hence, if the q-th frame of traffic stream i arrives at time a_i^q, it must wait for all frames from other streams with identical priority, which arrived before or at a_i^q, as well as wait for its own $q - 1$ predecessor frames to finish [21]:

$$I_i^{SPB}(q, a_i^q) = (q-1) C_i^+ + \sum_{j \in sp(i)} \eta_j^+(a_i^q + \epsilon) C_j^+ \tag{24.48}$$

Here, $sp(i)$ is a function yielding the set of all traffic streams whose priority is equal to that of stream i (excluding stream i). In the worst case, any same-priority frames arriving concurrently at exactly a_i^q are assumed to interfere with the q-frame of stream i. Again, an infinitesimal small time ϵ is added to Δt to cover this case.

In [21] it is shown that FIFO scheduling requires a candidate search in order to determine the worst-case blocking. The reason for this candidate search is that if frame q arrives early (within its jitter bounds), it might experience additional blocking from some of its own $q-1$ queued predecessors. However, if it arrives late (within its jitter bounds), it might experience additional blocking from previously queued frames of interfering same-priority streams. The set of arrival candidates A_i^q can be reduced to points in time where the candidates a_i^q coincide with the earliest arrivals of interfering frames from same-priority traffic streams [21]. Consequently, all candidates for the arrival of the q-th frame of stream i can be found by investigating the arrivals of interfering frames between the earliest arrival $\delta_i^-(q)$ of the q-th frame and its q-activation scheduling horizon $S_i(q)$, which is the time a switch port is busy processing q frames of stream i, including interfering frames from other traffic streams (cf. Eq. (4) in ▸ Chap. 23, "CPA: Compositional Performance Analysis"):

$$A_i^q = \bigcup_{j \in sp(i)} \left\{ \delta_j^-(n) | \delta_i^-(q) \leq \delta_j^-(n) < S_i(q) \right\}_{n \geq 1} \tag{24.49}$$

where, in the context of standard Ethernet, $S_i(q)$ can be computed as follows:

$$S_i(q) = \max_{j \in lp(i)} \left\{ C_j^+ \right\} + qC_i^+ + \sum_{j \in sp(i) \cup hp(i)} \eta_j^+ \left(S_i(q) \right) C_j^+ \tag{24.50}$$

Note that the computation of the q-activation scheduling horizon does not require a candidate search for the same-priority interference, as it is only concerned about the time when the port is busy. As $S_i(q)$ occurs on both sides, Eq. (24.50) cannot be solved directly. However, it represents an integer fixed-point problem, which can be solved by iteration, as all terms are monotonically increasing (cf. [29]). A valid starting point is, e.g., $S_i(q) = \max_{j \in lp(i)} \{C_j^+\} + qC_i^+$.

In order to compute the worst-case queuing delay $Q_i(q, a_i^q)$ of the q-th frame arrival of traffic stream i, which arrived at time a_i^q, all presented blocking effects must be considered:

$$Q_i(q, a_i^q) = I_i^{LPB} + I_i^{SPB}(q, a_i^q) + I_i^{HPB}\left(Q_i(q, a_i^q) \right) \tag{24.51}$$

Again, $Q_i(q, a_i^q)$ occurs on both sides, and Eq. (24.51) cannot be solved directly. Like the integer fixed-point problem in Eq. (24.50), it can be solved by iteration with, e.g., $Q_i(q, a_i^q) = (q-1)C_i^+$ as a starting point.

Now, the largest transmission latency $R_i(q)$ for the q-th frame arrival of traffic stream i can be computed by adding the transmission time C_i^+ of this q-th frame to its worst-case queuing delay and accounting for the fact that the frame arrived at time a_i^q (see, e.g., [3]). This is illustrated in Fig. 24.8:

$$R_i(q) = \max_{a_i^q \in A_i^q} \left\{ Q_i(q, a_i^q) + C_i^+ - a_i^q \right\} \tag{24.52}$$

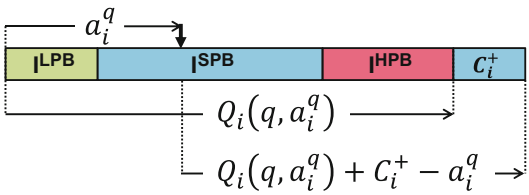


Fig. 24.8 Example queuing delay and transmission latency computation (cf. [75])

By taking the maximum overall $R_i(q)$, the worst-case frame transmission latency for a frame of stream i can be computed:

$$R_i^+ = \max_{1 \le q \le q_i^+} \{R_i(q)\} \tag{24.53}$$

As mentioned before, in order to derive the worst-case frame transmission latency of stream i, all frame arrivals of stream i in its longest scheduling horizon must be evaluated. Let q_i^+ be the maximum number of these frame arrivals. It can be derived by computing the maximum number of frames, which arrive during the scheduling horizon of their respective predecessors (cf. [18]):

$$q_i^+ = \max_{q \ge 1} \{q | \delta_i^-(q) \le S_i(q-1)\} \tag{24.54}$$

Now, as established in Sects. 2.2.1 and 2.2.2 in ▸ Chap. 23, "CPA: Compositional Performance Analysis", the worst-case bounds on the maximum path latency and the maximum frame backlog can be derived from the maximum frame transmission latencies and maximum q-activation processing times.

24.4.3.1 End-to-End Latency Bounds

From the individual worst-case transmission latencies of the frames along the path of a traffic stream through the network, the worst-case end-to-end latency of the stream can be derived. Let $Path(i)$ be the path of stream i through the network. Now, the time it takes to transmit q frames of stream i, i.e., its worst-case q-activation end-to-end latency, can be bounded by (cf. [20]):

$$L_i^+(q) = \delta_i^-(q) + \sum_{j \in Path(i)} R_j^+ \tag{24.55}$$

Here, the frames of stream i are injected into the network at their maximum rate (i.e., with minimum inter-arrival times $\delta_i^-(q)$) to induce maximum load on the system's resources. Along any given path, frames of a traffic stream are processed in order, i.e., they cannot overtake each other. Equation (24.55) assumes that the last of the q frames experiences the worst-case transmission latency on all its ports. Due to in-order processing, all $q-1$ previously sent frames must have arrived by then.

Obviously, for $q = 1$, Eq. (24.55) yields the worst-case end-to-end latency of a single frame (recall that $\delta_i^-(1) = 0$). Larger q are convenient in cases where, for example, a large IP packet is distributed over multiple Ethernet frames.

24.4.3.2 Buffer Size Bounds

Apart from timing guarantees, buffer size requirements are also important, as actual systems (e.g. ,switches) only have limited memory resources (buffer space). Insufficient buffer space can lead to frame drop, which is highly undesirable for (time) critical traffic.

The maximum activation backlog of a traffic stream i is an upper bound on the number of frames from i that can be queued at a resource at any given time. It can be derived by computing, for each q-th frame, the maximum number of frames, which arrived until the q-th frame has been transmitted, and subtracting from this number the $q - 1$ frames that must have been transmitted prior to the q-th one (cf. [21]):

$$b_i^+ = \max_{1 \le q \le q_i^+} \left\{ \eta_i^+ \left(B_i^+(q) \right) - q + 1 \right\} \tag{24.56}$$

where $B_i^+(q)$ is the multiple activation processing time. Given q consecutive frames of a traffic stream i, the multiple activation processing time is the longest time interval between the arrival of the first frame and end of the transmission of the q-th frame (cf. Eq. (8) in ▸ Chap. 23, "CPA: Compositional Performance Analysis"):

$$B_i^+(q) = Q_i(q) + C_i^+ \tag{24.57}$$

In the Ethernet context, it can be bounded by the multiple activation queuing delay under the assumption that all event arrive as soon as possible (i.e., we do not need to consider different event arrivals as in Eq. (24.51)) by adding the frames worst-case transmission time C_i^+ (cf. Eq. (9) in ▸ Chap. 23, "CPA: Compositional Performance Analysis"):

$$Q_i(q) = I_i^{LPB} + I_i^{SPB}(q, Q_i(q)) + I_i^{HPB}(Q_i(q)) \tag{24.58}$$

From the maximum activation backlogs, the maximum buffer size requirements can be derived. Typically, the memory in Ethernet switches can only be allocated block wise, e.g., in blocks of 128 or 256 bytes. This must be taken into account when deriving the maximum buffer size requirements. Assuming that a switch only allows the allocation of memory blocks of size m and that only the destination and source MAC addresses, the IEEE 802.1Q tag, the EtherType, the maximum payload p^+, and the frame check sequence of an Ethernet frame must be stored in switch memory, the buffer size requirement (in bytes) for a traffic stream i can be bounded by (cf. Sect. 24.4.2):

$$\hat{b}_i^+ = b_i^+ \left\lceil \frac{22\,\text{bytes} + \max\{42\,\text{bytes}, p^+\}}{m} \right\rceil m \tag{24.59}$$

The buffer size requirement per port can be computed by summing the buffer size requirements of all streams passing this port, and the buffer size requirement of a switch can be computed by summing up the requirements of each of its ports.

24.4.4 Analysis Extensions

24.4.4.1 Other Ethernet Schedulers

As Ethernet strives to cover a wide range of application domains, it supports many different schedulers and shapers to forward frames, each of which has a different impact on the queuing delay at a switch's output port. For the most prominent ones, CPA-based analyses are available.

Ethernet AVB [31] introduced standardized traffic shaping in the form of credit-based shaping on top of standard Ethernet. The motivation is to shape higher-priority traffic streams to bound their interference on lower-priority ones, e.g., to prevent starvation. However, as any form of traffic shaping introduces additional delays, a careful timing analysis is required to evaluate Ethernet AVB's applicability for real-time applications. Formal analyses for Ethernet AVB in the context of the CPA framework are presented in [21] and [3].

Ethernet TSN defines a set of Ethernet standards, which were designed with real-time requirements in mind. Some of these standards specify new link arbitration mechanisms. Namely, IEEE 802.1Qbv [35] introduces time-triggered frame forwarding to Ethernet, i.e., frames of time-triggered traffic classes are scheduled at predefined points in time such that they do not experience interference from other traffic classes. IEEE 802.1Qbv relies on so-called guard bands to block non-time-triggered traffic early enough to prevent interference with time-triggered traffic. In IEEE 802.1Qch [32], cyclic frame forwarding is defined, i.e., frame forwarding is based on alternating time intervals and frames received in one interval will be sent in the next interval etc. A new credit-based shaper, which aims to improve the forwarding of bursts, is discussed in [25]. Formal analyses for these shapers are presented in [75] and [74].

Although not explicitly standardized by the IEEE, weighted round robin scheduling can be implemented as an IEEE 802.1Q enhanced transmission selection algorithm. A CPA-compatible formal analysis for weighted round robin scheduling in the Ethernet context has been presented in [71].

In order to improve the timing of critical traffic, frame preemption has been introduced to Ethernet via the IEEE 802.3br [33] and IEEE 802.1Qbu [34] standards. A CPA-compatible formal analysis for frame preemption has been presented in [73].

24.4.4.2 Analysis Improvements

This section covered the fundamental approach to derive timing guarantees for Ethernet networks in CPA. The presented baseline analysis has been improved and extended in many directions.

Different analysis optimizations to exploit various kinds of correlations between Ethernet traffic streams have been proposed in [3] and [70]. Axer et al. [3] exploits

the fact that both Ethernet links and Ethernet AVB's traffic shapers limit the amount of workload, which can pass them in a given time interval. This property can be used to limit the interference during the computation of the worst-case frame transmission latencies. In [70], the authors show how FIFO scheduling can be exploited to reduce the interference a frame can experience from its same-priority predecessors.

24.4.4.3 Higher-Layer Protocols

Ethernet only defines frame forwarding on layer 2 of the ISO/OSI model. Higher-layer protocols often have additional timing implications. In [2] and [72] analyses to determine a bound on the worst-case timing impact of Automatic Repeat Requests (ARQs) and software-defined networking (SDN) [42] are presented.

Due to the compositional nature of CPA, the Ethernet analysis can be easily combined with other analyses from the CPA framework to derive system-wide performance guarantees. In [76], this has been done to compute end-to-end latency bounds for CAN-over-Ethernet traffic, where Ethernet ports are modeled as described in this section, but CAN buses and gateway processors are modeled according to their respective scheduling policies.

24.5 Conclusion

This chapter gives the overview on three representative communication protocols for real-time embedded systems, focusing on their timing related design principles and analysis. We note that real-time embedded systems are increasingly equipped with more sophisticated features (such as autonomous driving) that require high adaptivity and large volume data exchange. As a consequence, we envision that their future supporting communication networks will provide better extensibility and higher bandwidth while still keeping their behavior predictable.

Acknowledgments The contribution *Packet-Switched Networks: Ethernet* has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 644080.

References

1. Andersson B, Tovar E (2009) The utilization bound of non-preemptive rate-monotonic scheduling in controller area networks is 25%. In: 2009 IEEE international symposium on industrial embedded systems, pp 11–18
2. Axer P, Thiele D, Ernst R (2014) Formal timing analysis of automatic repeat request for switched real-time networks. In: Proceedings of the SIES, Pisa
3. Axer P, Thiele D, Ernst R, Diemer J (2014) Exploiting shaper context to improve performance bounds of Ethernet AVB Networks. In: Proceedings of the DAC, San Francisco
4. Baruah S, Chen D, Gorinsky S, Mok A (1999) Generalized multiframe tasks. Real-Time Syst 17(1):5–22

5. Broster I, Burns A, Rodriguez-Navas G (2002) Probabilistic analysis of can with faults. In: 23rd IEEE real-time systems symposium, pp 269–278
6. von der Bruggen G, Chen JJ, Huang WH (2015) Schedulability and optimization analysis for non-preemptive static priority scheduling based on task utilization and blocking factors. In: 2015 27th Euromicro conference on real-time systems (ECRTS). IEEE, pp 90–101
7. Casparsson L, Rajnak A, Tindell K, Malmberg P (1998) Volcano revolution in on-board communications. Technical report, Volvo
8. Chen Y, Kurachi R, Takada H, Zeng G (2011) Schedulability comparison for can message with offset: priority queue versus FIFO queue. In: 19th international conference on real-time and network systems, pp 181–192
9. Darbandi A, Kim MK (2014) Schedule optimization of static messages with precedence relations in FlexRay. In: Sixth international conference on ubiquitous and future networks, pp 495–500
10. Darbandi A, Kwon S, Kim MK (2014)Scheduling of time triggered messages in static segment of FlexRay. Int J Softw Eng Appl 8(6):195–208
11. Davis R, Navet N (2012) Controller area network (CAN) schedulability analysis for messages with arbitrary deadlines in FIFO and work-conserving queues. In: 9th IEEE international workshop on factory communication systems, pp 33–42
12. Davis RI, Burns A, Bril RJ, Lukkien JJ (2007) Controller area network (CAN) schedulability analysis: refuted, revisited and revised. Real-Time Syst 35(3):239–272
13. Davis RI, Kollmann S, Pollex V, Slomka F (2013) Schedulability analysis for controller area network (CAN) with FIFO queues priority queues and gateways. Real-Time Syst 49(1): 73–116
14. Di Natale M, Zeng H (2010) System identification and extraction of timing properties from controller area network (CAN) message traces. In: IEEE conference on emerging technologies and factory automation, pp 1–8
15. Di Natale M, Zeng H (2013) Practical issues with the timing analysis of the controller area network. In: 18th IEEE conference on emerging technologies factory automation, pp 1–8
16. Di Natale M, Zeng H, Giusto P, Ghosal A (2012) Understanding and using the controller area network communication protocol: theory and practice. Springer Science & Business Media, New York
17. Diemer J (To appear) Predictable network-on-chip for general-purpose processors – formal worst-case guarantees for on-chip interconnects. Ph.D. thesis, Technische Universität Braunschweig, Braunschweig. N/A
18. Diemer J, Axer P, Ernst R (2012) Compositional performance analysis in python with pyCPA. In: International workshop on analysis tools and methodologies for embedded and real-time systems
19. Diemer J, Rox J, Ernst R (2012) Modeling of Ethernet AVB networks for worst-case timing analysis. In: MATHMOD – Vienna international conference on mathematical modelling, Vienna
20. Diemer J, Rox J, Negrean M, Stein S, Ernst R (2011) Real-time communication analysis for networks with two-stage arbitration. In: Proceedings of the ninth ACM international conference on embedded software (EMSOFT 2011). ACM, Taipei, pp 243–252
21. Diemer J, Thiele D, Ernst R (2012) Formal worst-case timing analysis of ethernet topologies with strict-priority and AVB switching. In: IEEE international symposium on industrial embedded systems. Invited Paper
22. Ding S (2010) Scheduling approach for static segment using hybrid genetic algorithm in FlexRay systems. In: 10th IEEE international conference on computer and information technology, pp 2355–2360
23. Ding S, Murakami N, Tomiyama H, Takada H (2005) A ga-based scheduling method for FlexRay systems. In: 5th ACM international conference on embedded software, pp 110–113
24. Ghosal A, Zeng H, Di Natale M, Ben-Haim Y (2010) Computing robustness of FlexRay schedules to uncertainties in design parameters. In: Proceedings of the conference on design, automation and test in Europe, pp 550–555

25. Götz FJ (2013) Alternative shaper for scheduled traffic in time sensitive networks. In: IEEE 802.1 TSN TG meeting, Vancouver
26. Grenier M, Havet L, Navet N (2008) Configuring the communication on FlexRay-the case of the static segment. In: 4th European congress on embedded real time software
27. Han G, Di Natale M, Zeng H, Liu X, Dou W (2013) Optimizing the implementation of real-time simulink models onto distributed automotive architectures. J Syst Archit 59(10): 1115–1127
28. Han G, Zeng H, Li Y, Dou W (2014) SAFE: security-aware FlexRay scheduling engine. In: Design, automation and test in Europe conference and exhibition
29. Henia R, Hamann A, Jersak M, Racu R, Richter K, Ernst R (2005) System level performance analysis – the SymTA/S approach. In: IEE proceedings computers and digital techniques
30. Hu M, Luo J, Wang Y, Lukasiewycz M, Zeng Z (2014) Holistic scheduling of real-time applications in time-triggered in-vehicle networks. IEEE Trans Ind Inf 10(3): 1817–1828
31. IEEE Audio Video Bridging Task Group (2010) 802.1Qav – forwarding and queuing enhancements for time-sensitive streams. http://www.ieee802.org/1/pages/802.1av.html
32. IEEE Audio Video Bridging Task Group (2016) 802.1Qch – cyclic queuing and forwarding. http://www.ieee802.org/1/pages/802.1ch.html
33. IEEE P802.3br Interspersing Express Traffic Task Force. P802.3br – standard for ethernet amendment specification and management parameters for interspersing express traffic. https://standards.ieee.org/develop/project/802.3br.html
34. IEEE Time-Sensitive Networking Task Group. 802.1Qbu – frame preemption. http://www.ieee802.org/1/pages/802.1bu.html
35. IEEE Time-Sensitive Networking Task Group (2015) P802.1Qbv (Draft 3.0) – enhancements for scheduled traffic. http://www.ieee802.org/1/pages/802.1bv.html
36. International Standards Organisation (ISO) (1993) ISO 11898-1. Road vehicles – interchange of digital information – controller area network (CAN) for high-speed communication. ISO Standard-11898
37. International Standards Organisation (ISO) (2013) Road vehicles – FlexRay communications system – part 1: general information and use case definition. ISO Standard-17458
38. Jansen K, Solis-Oba R (2003) An asymptotic fully polynomial time approximation scheme for bin covering. Theor Comput Sci 306(1):543–551
39. Kang M, Park K, Jeong MK (2013) Frame packing for minimizing the bandwidth consumption of the FlexRay static segment. IEEE Trans Ind Electron 60(9):4001–4008
40. Khan D, Bril R, Navet N (2010) Integrating hardware limitations in can schedulability analysis. In: 8th IEEE international workshop on factory communication systems, pp 207–210
41. Khan D, Davis R, Navet N (2011) Schedulability analysis of can with non-abortable transmission requests. In: 16th IEEE conference on emerging technologies factory automation, pp 1–8
42. Kreutz D, Ramos F, Esteves Verissimo P, Esteve Rothenberg C, Azodolmolky S, Uhlig S (2015) Software-defined networking: a comprehensive survey. Proc IEEE 103(1):14–76
43. Li W, Di Natale M, Zheng W, Giusto P, Sangiovanni-Vincentelli A, Seshia S (2009) Optimizations of an application-level protocol for enhanced dependability in flexray. In: Design, automation test in Europe conference exhibition (DATE 2009), pp 1076–1081
44. Lincoln B, Cervin A (2002) Jitterbug: a tool for analysis of real-time control performance. In: Proceedings of the 41st IEEE conference on decision and control, vol 2, pp 1319–1324
45. Liu M, Behnam M, Nolte T (2013) An EVT-based worst-case response time analysis of complex real-time systems. In: 8th IEEE international symposium on industrial embedded systems, pp 249–258
46. Liu M, Behnam M, Nolte T (2013) Schedulability analysis of multi-frame messages over controller area networks with mixed-queues. In: 18th IEEE conference on emerging technologies factory automation, pp 1–6
47. Liu M, Behnam M, Nolte T (2014) Schedulability analysis of GMF-modeled messages over controller area networks with mixed-queues. In: 10th IEEE workshop on factory communication systems, pp 1–10

48. Lukasiewycz M, Glaß M. Teich J, Milbredt P (2009) FlexRay schedule optimization of the static segment. In: 7th IEEE/ACM international conference on hardware/software codesign and system synthesis, pp 363–372
49. Lukasiewycz M, Schneider R, Goswami D, Chakraborty S (2012) Modular scheduling of distributed heterogeneous time-triggered automotive systems. In: 17th Asia and South Pacific design automation conference, pp 665–670
50. Mok A, Chen D (1996) A multiframe model for real-time tasks. In: 17th IEEE real-time systems symposium, pp 22–29
51. Mubeen S, Mäki-Turja J, Sjödin M (2011) Extending schedulability analysis of controller area network (CAN) for mixed (periodic/sporadic) messages. In: 16th IEEE conference on emerging technologies factory automation, pp 1–10
52. Mubeen S, Mäki-Turja J, Sjödin M (2012) Extending response-time analysis of mixed messages in can with controllers implementing non-abortable transmit buffers. In: 17th IEEE conference on emerging technologies factory automation, pp 1–4
53. Mubeen S, Mäki-Turja J, Sjödin M (2012) Response time analysis for mixed messages in can supporting transmission abort requests. In: 7th IEEE international symposium on industrial embedded systems, pp 291–294
54. Mubeen S, Mäki-Turja J, Sjödin M (2012) Response-time analysis of mixed messages in controller area network with priority- and FIFO-queued nodes. In: 9th IEEE international workshop on factory communication systems, pp 23–32
55. Mubeen S, Mäki-Turja J, Sjödin M (2012) Worst-case response-time analysis for mixed messages with offsets in controller area network. In: 17th IEEE conference on emerging technologies factory automation, pp 1–10
56. Mubeen S, Mäki-Turja J, Sjödin M (2013) Extending offset-based response-time analysis for mixed messages in controller area network. In: 18th IEEE conference on emerging technologies factory automation, pp 1–10
57. Mubeen S, Mäki-Turja J, Sjödin M (2014) Extending worst case response-time analysis for mixed messages in controller area network with priority and FIFO queues. IEEE Access 2: 365–380
58. Mubeen S, Mäki-Turja J, Sjödin M (2014) Response time analysis with offsets for mixed messages in can supporting transmission abort requests. In: Emerging technology and factory automation (ETFA 2014). IEEE, pp 1–10
59. Mubeen S, Mäki-Turja J, Sjödin M (2015) Integrating mixed transmission and practical limitations with the worst-case response-time analysis for controller area network. J Syst Softw 99:66–84
60. Mundhenk P, Steinhorst S, Lukasiewycz M, Fahmy SA, Chakraborty S (2015) Security analysis of automotive architectures using probabilistic model checking. In: 52nd ACM/IEEE design automation conference (DAC), pp 1–6
61. Natale MD (2006) Evaluating message transmission times in controller area networks without buffer preemption. In: 8th Brazilian workshop on real-time systems
62. Navet N, Song YQ, Simonot F (2000) Worst-case deadline failure probability in real-time applications distributed over controller area network. J Syst Archit 46(7):607–617
63. Neukirchner M, Negrean M, Ernst R, Bone TT (2012) Response-time analysis of the FlexRay dynamic segment under consideration of slot-multiplexing. In: 7th IEEE international symposium on industrial embedded systems, pp 21–30
64. Nolte T, Hansson H, Norstrom C (2003) Probabilistic worst-case response-time analysis for the controller area network. In: 9th IEEE real-time and embedded technology and applications symposium, pp 200–207
65. Pop T, Pop P, Eles P, Peng Z, Andrei A (2008) Timing analysis of the FlexRay communication protocol. Real-Time Syst 39(1–3):205–235
66. Schenkelaars T, Vermeulen B, Goossens K (2011) Optimal scheduling of switched FlexRay networks. In: Design, automation test in Europe conference exhibition, pp 1–6
67. Schmidt K, Schmidt E (2009) Message scheduling for the FlexRay protocol: the static segment. IEEE Trans Veh Technol 58(5):2170–2179

68. Tanasa B, Bordoloi UD, Kosuch S, Eles P, Peng Z (2012) Schedulability analysis for the dynamic segment of FlexRay: a generalization to slot multiplexing. In: 18th IEEE real-time and embedded technology and applications symposium, pp 185–194
69. Tanasa B, Dutta Bordoloi U, Eles P, Peng Z (2011) Reliability-aware frame packing for the static segment of FlexRay. In: Proceedings of the ninth ACM international conference on embedded software, pp 175–184
70. Thiele D, Axer P, Ernst R (2015) Improving formal timing analysis of switched ethernet by exploiting FIFO scheduling. In: Design automation conference (DAC), San Francisco
71. Thiele D, Diemer J, Axer P, Ernst R, Seyler J (2013) Improved formal worst-case timing analysis of weighted round robin scheduling for ethernet. In: Proceedings of the CODES+ISSS, Montreal
72. Thiele D, Ernst R (2016) Formal analysis based evaluation of software defined networking for time-sensitive ethernet. In: Proceedings of the design, automation, and test in Europe (DATE), Dresden
73. Thiele D, Ernst R (2016) Formal worst-case performance analysis of time-sensitive Ethernet with frame preemption. In: Proceedings of emerging technologies and factory automation (ETFA), Berlin, p 9
74. Thiele D, Ernst R (2016) Formal worst-case timing analysis of Ethernet TSN's burst-limiting shaper. In: Proceedings of the design, automation, and test in Europe (DATE), Dresden
75. Thiele D, Ernst R, Diemer J (2015) Formal worst-case timing analysis of Ethernet TSN's time-aware and peristaltic shapers. In: IEEE vehicular networking conference (VNC)
76. Thiele D, Schlatow J, Axer P, Ernst R (2015) Formal timing analysis of can-to-ethernet gateway strategies in automotive networks. Real-Time Syst. http://dx.doi.org/10.1007/s11241-015-9243-y
77. Tindell K, Hansson H, Wellings A (1994) Analysing real-time communications: controller area network (CAN). In: IEEE real-time systems symposium, pp 259–263
78. Vector. CANbedded interaction layer. [Online] http://www.vector.com
79. Yomsi P, Bertrand D, Navet N, Davis R (2012) Controller area network (CAN): response time analysis with offsets. In: 9th IEEE international workshop on factory communication systems, pp 43–52
80. Zeng H, Di Natale M, Ghosal A, Sangiovanni-Vincentelli A (2011) Schedule optimization of time-triggered systems communicating over the FlexRay static segment. IEEE Transactions on Industrial Informatics 7(1):1–17
81. Zeng H, Di Natale M, Giusto P, Sangiovanni-Vincentelli A (2009) Stochastic analysis of CAN-based real-time automotive systems. IEEE Transactions on Industrial Informatics 5(4):388–401
82. Zeng H, Di Natale M, Giusto P, Sangiovanni-Vincentelli A (2010) Using statistical methods to compute the probability distribution of message response time in controller area network. IEEE Transactions on Industrial Informatics 6(4):678–691
83. Zeng H, Ghosal A, Di Natale M (2010) Timing analysis and optimization of FlexRay dynamic segment. In: 7th IEEE international conference on embedded software and systems, pp 1932–1939

Part VII
Hardware/Software Compilation and Synthesis

Hardware-Aware Compilation

25

Aviral Shrivastava and Jian Cai

Abstract

Hardware-aware compilers are in high demand for embedded systems with stringent multidimensional design constraints on cost, power, performance, etc. By making use of the microarchitectural information about a processor, a hardware-aware compiler can generate more efficient code than a generic compiler while meeting the design constraints, by exploiting those highly customized microarchitectural features. In this chapter, we introduce two applications of hardware-aware compilers: a hardware-aware compiler can be used as a production compiler and as a tool to efficiently explore the design space of embedded processors. We demonstrate the first application with a compiler that generates efficient code for embedded processors that do not have any branch predictor to reduce branch penalties. To demonstrate the second application, we show how a hardware-aware compiler can be used to explore the Design Space of the bypass designs in the processor. In both the cases, the hardware-aware compiler can generate better code than a hardware-ignorant compiler.

Acronyms

ADL	Architecture Description Language
BRF	Bypass Register File
BTB	Branch Target Buffer
CFG	Control-Flow Graph
CIL	Compiler-In-the-Loop
DSE	Design Space Exploration

A. Shrivastava (✉)
School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA
e-mail: aviral.shrivastava@asu.edu

J. Cai
Arizona State University, Tempe, AZ, USA
e-mail: jian.cai@asu.edu; jcai19@asu.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_26

HPC Horizontally Partitioned Cache
ISA Instruction-Set Architecture
MAC Multiply-Accumulator
OT Operation Table
RT Response Time
SPU Synergistic Processor Unit

Contents

25.1 Introduction

Hardware-aware compilation refers to the compilation that exploits the microarchitectural information of the processor to generate better code. Minimally, compilers only require information about the Instruction-Set Architecture (ISA) of the processor to generate code. This ISA-dependent compilation is often good-enough to generate code for high-performance superscalar processors, in which the hardware may drastically modify the instruction stream (e.g., break complex instructions into simpler microinstructions, fuse simple instructions into complex macro-instructions, reorder the instruction execution, and perform speculative and predictive computations) for efficient execution.

However, the processors in embedded systems, or embedded processors, are characterized by lean designs and specialization for the application domain [12, 16]. To meet the strict multidimensional constraints of the embedded systems, customization is very important. For example, even though register renaming improves performance in processors by avoiding false data dependencies, embedded processors may not be able to employ it because of the high power consumption and the increased complexity of the logic. Therefore embedded processors might deploy a "trimmed-down" or "lightweight" version of register renaming, for example, register scoreboarding, which provides a different trade-off in the cost, complexity, power, and performance of the embedded system. In addition, designers often implement some irregular design features, which are not common in general purpose processors, but will lead to significant improvements in some design parameters for the relevant set of applications. For example, several cryptography application processors come with hardware accelerators that implement the complex cryptography algorithm in the hardware. By doing so, the cryptography applications can be made faster, and consume less power, but may not have any noticeable impact on normal applications. Embedded processor architectures often have such application-specific "idiosyncratic" architectural features. And last but not the least, some design features that are present in the general-purpose processors may be

entirely missing in embedded processors. For example, support for prefetching is now a standard feature in general-purpose processors, but it may consume too much energy and require too much extra hardware to be appropriate in an embedded processor.

How can we effectively compile for such uniquely designed embedded processors? Just the information about the ISA is not enough. A good uniquely designed compiler needs microarchitectural information, including sizes of caches, buffers, and execution policies (e.g., register scoreboarding, branch prediction mechanism, etc). Many of these microarchitectural features are independent of the ISA but affect the performance very significantly [20, 28]. By knowing about these microarchitectural features, compilers can design a plan for efficient execution. For example, popular compilers such as GCC [13] and Clang [19], inlines many functions and unrolls loops to improve the run-time performance of applications. However, these optimization techniques increase program code size and thus may not be usable for embedded systems that have very limited instruction memory. To accommodate such diversified and sometimes multidimensional design restraints, not only the compiler must be aware of the memory size of the processor but also make sure that the compiled code can reside in the available memory. A compiler that uses microarchitectural information to generate efficient code is called a hardware-aware compiler.

Figure 25.1 shows the general flow of a hardware-aware compiler. The architectural description is provided along the input program to the compiler, in an Architecture Description Language (ADL). An ADL is a formal language that is used to describe the architecture of a system, including the memory hierarchy, pipeline stages, etc. Examples of ADLs include EXPRESSION [11], LISA [34], and RADL [30]. By taking into consideration the microarchitectural features described by the ADL, the compiler can generate a code that is better optimized for the target architecture. For example, by taking into consideration the memory access timing, the compiler may be able to generate better schedules of instructions for execution [8, 9].

Clearly a hardware-aware compiler is valuable as a production compiler, where it is used to generate carefully tuned code for the target microarchitecture, but it is

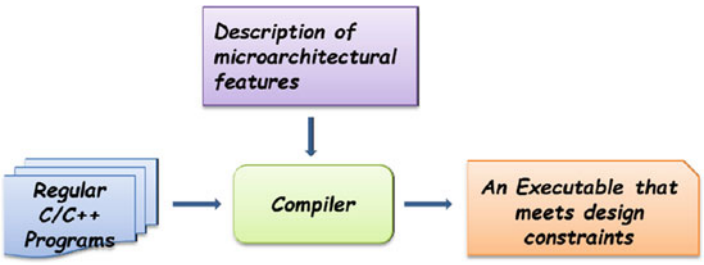


Fig. 25.1 Hardware-aware compiler uses the microarchitecture description of the processor to generate efficient code

also (and arguably even more) valuable for the design of the embedded processor itself. The typical way to design embedded processors, i.e., to determine the microarchitectural configuration – which microarchitectural features to keep in the processor, which execution policy, what buffer, which cache sizes, etc. – is through a simulator-based DSE. In this methodology, a cycle-accurate simulator of the processor with different microarchitectural features is designed. The applications are executed on the simulator to figure out which microarchitectural configuration works best. This methodology relies solely on processor simulators for Design Space Exploration (DSE) on traditional superscalar processors, since the code quality is not very important in them. However, including compiler in the DSE loop is more important for efficient designs of embedded processors, where a compiler can have very significant impact on the eventual power and performance of the application. In the Compiler-In-the-Loop (CIL) DSE methodology, the compiler is used to generate a code for each microarchitectural variation, and the best design is selected. This methodology enables us to pick microarchitectural configurations that may not be as effective by themselves, but lend themselves to very effective use by the compiler, and achieve superior power saving and performance improvement. Such configurations will be disregarded by the traditional simulation-only DSE.

In summary, hardware-aware compilers that use the microarchitectural information about the processor to generate better code can improve the power consumption and performance of embedded processors. They can be used both as a production compiler and be used in the compiler-based processor design. We dive into both of these uses of the hardware-aware compiler in the rest of this chapter. In the next section, we describe the use of a hardware-aware compiler as a production compiler. We present an example of a compiler that generates good code for embedded processors that do not have any branch predictor at all. Branch predictors, though very useful in eliminating most of the branch penalty, are costly (require a lot of hardware) and drain significant amount of power. As a result, some embedded processors may choose to drop them. Note that the presence/absence of a branch predictor does not affect the ISA but has a very significant impact on the power consumption and performance of the execution. Indeed without a branch predictor, all branches will incur branch penalty, and this will be excessive for execution. However, a compiler can help. Instead of expensive branch predictors, embedded processors may choose to have a branch hint instruction, which can indicate to the processor the direction of the imminent branch. If the application developer or the compiler can insert these branch hint instructions at the right places in the code, it can ameliorate most of the branch penalty and result in efficient execution.

Next we will describe how a hardware-aware compiler can be used as an effective tool in the design of embedded processors. We explain this through the example of designing the bypasses in the processor. In pipelined processors, even though the result is evaluated, it cannot be read by the next instruction (if there are no bypasses), until the instruction is committed, and writes its results in the register file. This pipeline penalty due to data dependencies among the instructions can be alleviated to a large extent by using bypasses that forward the results of instructions after evaluation to the operand read stage of dependent instructions. However, processors

now feature extremely long pipelines, often more than 20 stages [3, 6], and a full bypassing, e.g., bypasses from all the later stages of the pipeline to the operand read stage, can be extremely complex and expensive. Skipping some bypasses can reduce the overhead of bypassing logic. However, which bypasses to remove? Clearly, the bypasses that are used least often can be removed, but the compiler has an important role to play in this. If the compiler can reschedule the instructions around the missing bypasses, then the effect of the missing bypasses can be eliminated. The following section first describes, given a partial bypass configuration (i.e., not all bypasses are present), how do we reschedule the code so as to avoid the missing bypasses and then shows the results of DSE with and without this bypass-sensitive compiler in the loop.

For readers that are interested to learn more about related topics, ▶ Chap. 26, "Memory-Aware Optimization of Embedded Software for Multiple Objectives" introduces compiler-based techniques that map applications to embedded systems with scratchpad memories, focusing on minimizing the worst-case execution time of the applications. ▶ Chapter 27, "Microarchitecture-Level SoC Design" presents typical system-on-chip design flow and detailed issues in power modelings, thermal, and reliability, as well as their relation, and presented some interesting solutions.

25.1.1 Hardware-Aware Compilers as Production Compilers

A hardware-aware compiler can be used as a production compiler to generate a code for embedded systems once the microarchitecture is fixed. Researchers have discovered several use-cases for hardware-aware compilers. Muchnick [23] has developed the concept of Response Time (RT) and RT-based compiler that reschedules instructions to minimize the data dependence penalty in processors. A RT specifies how an operation may use the resources of a processor as the operation executes. Their compiler uses the specification of the pipeline of the processor as an input. Using this, it can create RT for the given instructions and detect conflicts among them – the structural and data hazards – so as to generate better instruction scheduling [21, 31]. Bala and Rubin [1] and Proebsting and Fraser [26] proposed compiler techniques that use the finite-state automaton (FSA), a derivative of the RT, to further speed up the detection of pipeline hazards during instruction scheduling. These approaches improve the power and performance of execution.

Hardware-aware compilers have also been proposed to help hide memory latency [25]. The most important source of memory latency in processors is the cache miss penalty, as cache misses typically take orders of magnitude longer time than cache hits. Grun et al. developed a compiler optimization that uses accurate timing information of both memory operations and the processor pipeline to exploit memory access modes, such as page mode and burst mode, so as to allow the compiler to reorder memory operations to help hide the memory latency [8]. They later extended the work and used memory access timing information to perform aggressive scheduling of memory operations, so that cache miss transfers can be overlapped with the cache hits and CPU operations [9]. For example, an instruction

that will cause a cache miss (known by cache analysis) can be scheduled earlier so that the following cache hits to the same cache line will not be stalled while the cache line is being transferred.

Hardware-aware compilers have also been proposed to reduce the power and temperature. Power gating [17, 24, 27] is one such application used in integrated circuit design to reduce the leakage power of processors. Leakage power already contributes to more than 30% of the power consumed by the processor. But by turning off the unused blocks, leakage power of that block can be reduced. However, power gating will backfire if the power spent in turning off and turning on an execution unit is more than the power saved while it is power gated or if we turn on the block too late, and there is a performance penalty corresponding to it. As a result, prediction-based techniques to power-gated blocks are not as effective. However, a compiler can analyze the application and find out regions of code where a functional block is not going to be used. If the functional block is going to be unused longer than a threshold of time, it can be safely power gated to minimize the leakage power of the functional block. To prevent such undesired leakage, the compiler can be utilized to analyze the control flow graph to predict the idle cycles of the execution units and ensure that power gating is applied only if the power saved during these cycles is greater than the power used to turn on/off the execution unit.

Another example of hardware-aware compilation to reduce power consumption can be found in computer architectures with Horizontally Partitioned Cache (HPC). An HPC architecture maintains multiple caches at the same level of memory hierarchy (in contrast to one cache per level to traditional computer architectures). Thanks to caching different kinds of data in separate caches to avoid interference between each other, e.g., between scalar variables and arrays, the HPC architecture is able to reduce the number of cache misses, which directly translates to the improved performance and abated power consumption. Moreover, HPC architectures include at the same level of memory hierarchies one or more small additional caches, aside the large-sized main cache. For example, in the Intel XScale [15], the L1 caches consist of the 32 KB main cache and a 2 KB additional cache. The additional caches typically consume less power per access, which further decreases the power consumption. Although the benefits of the HPC architecture are inviting, it is nontrivial to exploit such an architecture as its performance is highly dependent on the design parameters. Compiler techniques can be used to explore these parameters and carefully partition data to achieve the maximum benefit. For example, Shrivastava et al. identified the access pattern of data, and cached data with temporal locality in the main cache, while leaving data with spatial locality to the additional caches [29]. This is because the size of a cache does not affect the miss rate of memory accesses to data that exhibits spatial locality, while on the other hand, a larger (main) cache is able to have a higher chance to retain the data that shows temporal locality for repeated accesses.

For the rest of this subsection, we will present and detail a software branch hinting technique for processors without hardware branch prediction, but a simple software branch hinting mechanism [22].

25.1.1.1 The Case for Software Branch Hinting

Control hazards or branching hazards pose a serious limitation on the performance of pipelined processors, which becomes worse as the pipeline depth grows. A branch predictor that predicts the direction (taken or not taken) and the target address if the branch is to be taken can solve this predicament. Branch predictors are typically implemented in hardware so as to handle dynamism of branches. However, branch predictors can be expensive in both the area and power [4, 18]. As multi-core processors become increasingly popular even in embedded systems, some embedded multi-core processor designers remove the hardware branch predictors to meet the power cap while still being able to accommodate more cores. The IBM Cell processor [5], in an effort to improve its power efficiency, removes the hardware branch predictors from its Synergistic Processor Unit (SPU) coprocessors.

Doubtlessly, the lack of branch prediction will cause significant performance penalty. Table 25.1 manifests the huge overhead caused by branches when running some typical embedded benchmarks due to the lack of hardware branch prediction. To prevent such extreme performance loss, processors without hardware branch prediction may provide instructions for software branch prediction or software branch hinting, as the IBM Cell processor does. Branch hint instructions must be used wisely in such processors, in order to achieve comparable or even better performance than hardware branch prediction.

A branch hint instruction typically predicts the target address a branch will jump to when the branch is actually taken. This implies that such an instruction must be inserted only if it is for sure that the branch will be taken, to avoid the misprediction from slowing down the program execution. Fortunately, there have been many research works for predicting the direction of a branch [2, 32, 33] with pretty good accuracy. However, even if we know a branch is taken, finding an appropriate place in the program to insert the branch hint instruction is a nontrivial task. On the one hand, it takes time to set up the branch hint instruction, so the hint must be executed early enough to be recognized by the branch to be hinted. In other words, the branch hint instruction must be inserted early enough before the branch to take effect. On the other hand, there is also the restriction on the number of branch hint instructions that can be activated at the same time. Therefore, simply bringing forward the insertion point of a hint may cause problems in some cases. For example, in the IBM Cell processor, only one active branch hint instruction is allowed. So if there are two branches close to each other in the program, placing both hints of the two branches before the first branch (i.e., the second hint is also placed before the first branch to ensure there is enough time left for the second

Table 25.1 The percentage of execution time spent in branch penalty of typical embedded applications without branch prediction in Cell SPU

Benchmark	Branch penalty (%)
cnt	58.5
insert_sort	31.4
janne_complex	62.7
ns	50.9
select	36.2

branch to be hinted) may cause the effect of the first hint overwritten by the second hint, if the second hint has been activated by the time the first branch is executed. This will cause the misprediction at the first branch, force the execution to stall, and wait for the target address to be recalculated.

While the insertion of branch hint instructions can be done by programmers manually, it can be a tedious and time-consuming process. A compiler-based solution may be preferred. In the rest of this subsection, we will present a compiler-based approach for minimizing the branch penalty in processors with only software branch prediction. We will first introduce the model used for the cost function of branch penalties. It is based on the number of cycles between the hint instruction and the branch instruction, the taken probability of the branch, and the number of times the branch is executed. Subsequently, three basic methods for reducing branch penalties using the branch hint instruction are introduced and detailed:

(i) A no operation (NOP) padding scheme that inserts NOP instructions before a branch to leave enough time interval for its hint to be set up, for small basic blocks without enough margin originally.
(ii) A hint pipelining technique that allows two very close branches where originally only one of them can be hinted, to be now both hinted.
(iii) A loop restructuring technique that changes the loop structure so the compiler can insert the hints for more branches within the loop. The heuristic that combines and applies these basic methods to the code prudently is also briefly explained. Finally, experimental results collected are examined to demonstrate the efficacy of the technique.

The discussion will be based on the SPU coprocessor in the IBM Cell processor. However, the presented technique is applicable to other processors with only software branch prediction. Also, we will assume that every instruction takes one cycle for the sake of simplicity, although this is not necessarily true.

25.1.1.2 Mechanism of Software Branch Hinting

Figure 25.2 shows the overview of how a hint instruction works. The execution of a hint instruction comprises two stages: (i) launching the operation and setting up and (ii) loading the target instruction. Similar to hardware branch predictors, software branch hinting employs a Branch Target Buffer (BTB) to predict the target of a taken branch. When a hint instruction is executed, it needs to search the BTB and see if it can find any matched entry, and update the BTB if it fails to find one. This is done in the first stage. Once this stage is over, the hint instruction will start to fetch the target instruction into the hint target buffer. By default, the next instruction will be loaded to the in-line prefetch buffer, so the processor can fetch the instruction and continue the execution. However, when a branch instruction is identified, its PC address is used to search for any matching BTB entry (the BTB would have been updated, in the presence of a hint). If any entry is found matched, the processor will then instead fetch the next instruction from the hint target buffer.

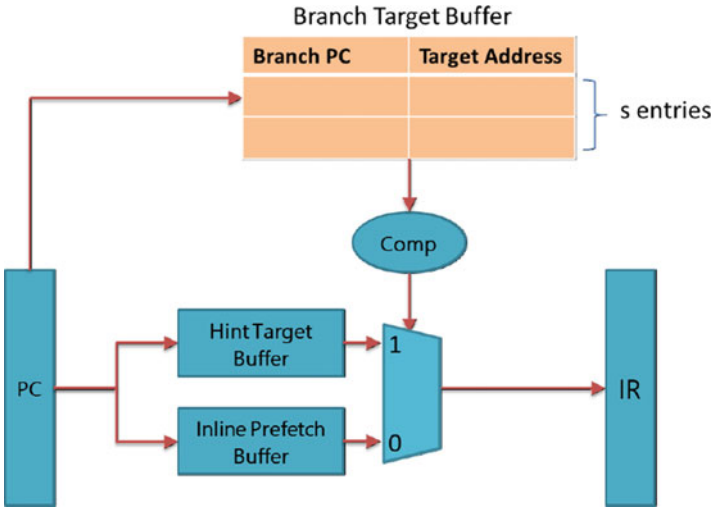


Fig. 25.2 The overview of software branch hint instructions

Given such mechanism of software branch hinting, we should readily identify the three critical design parameters that will seriously affect the performance of resultant implementation:

- d: the number of cycles used for starting up the operation
- f: the number of cycles to load the target instruction
- s: the number of BTB entries

The parameter d decides the minimum interval between a hint and the branch it aims to hint. In other words, a hint instruction must be executed at least d cycles earlier than the branch instruction in the program for it to take effect. After the startup of the hint instruction, a request is made to the arbiter [14] to load the target instruction from main memory into the hint target buffer. This is because in the cell processor, SPUs cannot access the main memory directly, so code and data must be first loaded into the local storage. This stage will take f cycles to complete. Once this stage is finished, the hint instruction is also completed. Therefore, if the hint instruction is executed $d + f$ cycles earlier, the branch it hints can be executed without any stall. In particular, if the branch is actually not taken, it will still wait for the hint instruction to load the incorrect target instruction, and then start over to load the correct instruction.

The number of BTB entries, s, decides size of the hint target buffer, since the buffer must be large enough to hold the target instructions for all the active hint instructions. For example, each SPU of the cell processor has only one entry in the BTB. Therefore, at the same time, only one active hint instruction is allowed for applications run on SPU. The bigger s is, the larger the BTB, and the more power consumption. Therefore, s is usually small.

25.1.1.3 Cost Model of Branch Penalties Under Software Branch Hinting

The branch penalty of a branch with software branch hinting can be modeled as the expected value of the penalty when the branch is successful predicted and when it is mispredicted, respectively. From our previous discussion of the software branch hinting mechanism, we know the branch penalty is related to the number of cycles between a hint and the branch to be hinted, whether the branch is correctly predicted. Therefore, the branch penalty can be modeled as below:

$$Penalty(l, n, p) = Penalty_{correct}(l) \times np + Penalty_{incorrect}(l) \times n(1-p) \quad (25.1)$$

where l, n, and p, respectively, represent the number of cycles between the hint and the branch, the number of times the branch is executed, and the branch probability. We assume n and p are given in our discussion. To find out the relation between the branch penalty and l when a branch is predicted correctly ($Penalty_{correct}(l)$) or incorrectly ($Penalty_{incorrect}(l)$), a synthetic benchmark that includes only a branch hint instruction and the branch instruction to hint is run in the SPU in the IBM cell processor. The hint and the branch are separated by *lnop* instructions (one type of NOP instructions). By increasing the number of *lnop* instructions between the hint and the branch, the change of branch penalties can be inferred through the variation of the execution time.

Two types of NOP instructions are available in the SPU, thanks to its dual-issue nature – *nop* for the *even* pipeline and *lnop* for the *odd* pipeline. The *even* pipeline is used to execute fixed point and floating point arithmetic operations, while the *odd* pipeline is used to execute memory, logic, and flow-control instructions, which include the branch instruction and the branch hint instruction. By filling only *lnop* and branch/hint instructions, the SPU is forced to use the *odd* pipeline only and issue one instruction at a time.

Figure 25.3 shows the relation between the branch penalty and the number of cycles between the hint instruction and the branch instruction, when the branch

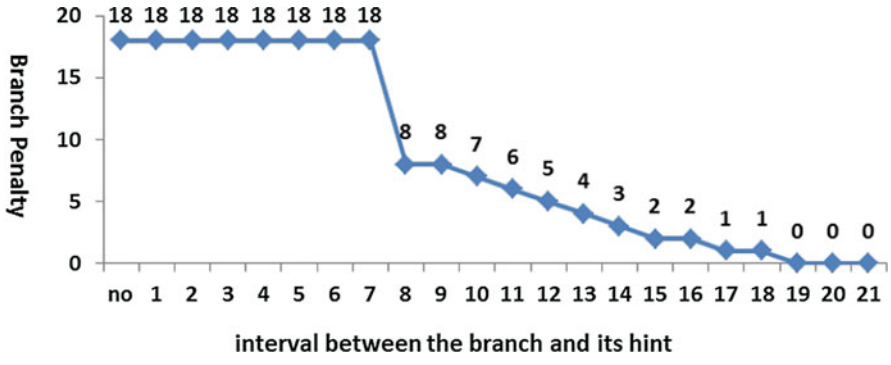


Fig. 25.3 The relation of the branch penalty, and the number of cycles between the hint instruction and the branch instruction, when the branch is predicted correctly

is correctly predicted. In other words, the branch is actually taken (since a hint instruction always loads the instruction at the target address of the taken branch). When the hint instruction is scheduled less than 8 cycles before the branch, the branch penalty is always 18 cycles. It implies the hint needs so much time to be properly set up and recognized when the branch starts to execute. **By default, the SPU always does not take branch prediction**. Therefore, without the hint, the SPU will keep predicting the incorrect direction for the branch and force the execution to pay the full branch penalty, i.e., resolving the target address and loading the instruction. The full branch penalty is measured as 18 cycles. As the interval between the hint and the branch is increased to be equal or greater than eight cycles (by inserting *lnop* instructions), the branch instruction is now aware of the existence of the hint. It still takes 18 cycles from the beginning of the hint instruction to the end of the branch instruction, since it also needs to resolve the branch target address and load the instruction, just like a branch instruction. However, by starting the entire process earlier, the hint instruction can hide some of the penalty, thanks to the instructions between the hint and the branch (the *lnop* instructions inserted), when the branch starts to execute. Notice the *lnop* instructions inserted are just placeholders for investigating the effect of the interval (between the hint and the branch) on the branch penalty. In the real execution, these will be replaced by the meaningful instructions. When the interval becomes equal or greater than 19 cycles, the branch penalty is completely eliminated. The branch penalty model in the SPU can be therefore built from the observation from the above experiment as follows:

$$Penalty_{correct}(l) \approx \begin{cases} 18, & if\ l < 8 \\ 18 - l, & if\ 8 \leq l < 19 \\ 0, & if\ l \geq 19 \end{cases} \tag{25.2}$$

where l denotes the number of cycles between the hint and the branch to be hinted.

Figure 25.4 shows the relation between the branch penalty and the number of cycles between the hint instruction and the branch instruction, when the branch is

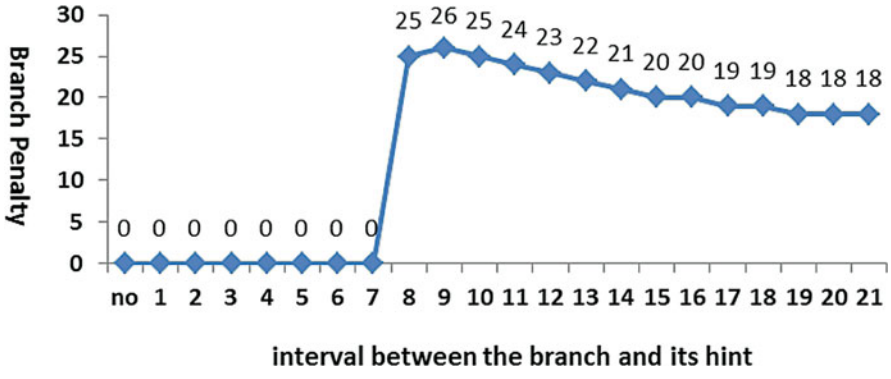


Fig. 25.4 The relation of the branch penalty, and the number of cycles between the hint instruction and the branch instruction, when the branch is mispredicted

mispredicted. In other words, the branch is not taken. When the interval is less than 8 cycles, the hint is not recognized at the branch, and it pays 18 cycles branch penalty just like before. However, when the interval is increased to be equal or greater than 8 cycles, the number of cycles spent from the beginning of the hint until the finish of the branch becomes greater than 18 cycles. This is because the branch instruction will start to wait for the incorrect target instruction to be loaded once it perceives the hint instruction. After the loading of the (incorrect) target instruction is completed, the branch instruction will start over, which will spend another 18 cycles. If the interval is further increased to be equal to or more than 19 cycles, the penalty becomes 18 cycles again, since the effect of misprediction will have been completely hidden by the time spent on executing the instructions between the hint and the branch, so that the branch will be executed as if the hint never happens and pays the 18 cycle penalty as if there is not any hint. The penalty of a branch misprediction thus can be modeled as follows:

$$Penalty_{incorrect}(l) \approx \begin{cases} 0, & if\ l < 8 \\ (18 - l) + 18 = 36 - l, & if\ 8 \leq l < 19 \\ 18, & if\ l \geq 19 \end{cases} \tag{25.3}$$

where l denotes the number of cycles between the hint and the branch to be hinted in the number of cycles.

25.1.1.4 Branch Hinting-Based Compilation

No Padding. When the number of cycles between the hint and the branch it will hint is smaller than a threshold value (eight in the SPU), the branch has to pay for the full branch penalty. In this case, we can insert NOP instructions (both *nop and lnop* instructions) to create a sufficiently large interval. Take Fig. 25.5 as an example.

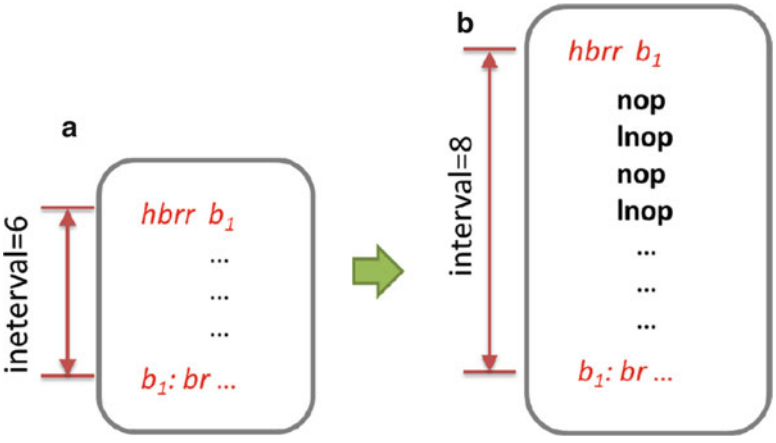


Fig. 25.5 NOP padding increases the interval between the hint instruction and the branch from 6 instructions/cycles in (**a**) to 8 instructions/cycles in (**b**) so that $b1$ can be hinted

Assume the branch *br* is taken and the hint instruction *hbrr* was originally executed six cycles earlier than *br*. According to the branch penalty model, the branch penalty is 18 cycles. After inserting two pairs of *nop/lnop* instructions, the interval becomes eight cycles, since each pair of *nop/lnop* can be executed in the *even/odd* pipeline at the same cycle, respectively, in the SPU. The branch penalty is therefore reduced to 10 cycles. The overall improvement is hence $18 - 10 - 2$ cycles, i.e., 6 cycles.

The SPU GCC compiler also provides a scheme for inserting *nop* (inserts both *nop* and *lnop* instruction) when a user-specified flag is enabled [7]. The SPU GCC inserts whenever the interval between a hint and the branch is not long enough. Carrying out NOP padding without deliberation may hurt performance sometimes.

To find out whether NOP padding is necessary under certain circumstances, we need a way to estimate the effect to the performance of applications if we insert the NOP instructions. Let l, n, p, and n_{NOP}, respectively, denote the interval between a hint and the branch to hint before NOP padding, the number of times the branch is executed, the taken probability of the branch, and the number of NOP instructions inserted. The branch penalties before the NOP padding can be calculated as follows:

$$Penalty_{no\_pad} = Penalty(l, n, p) \tag{25.4}$$

The branch penalties after the NOP padding can be calculated as follows,

$$Penalty_{pad} = Penalty(l + n_{NOP}, n, p) \tag{25.5}$$

For example, before the NOP padding, if the interval is smaller than eight cycles so that the hint is not recognized, then by applying the model of branch penalty introduced, we can get the branch penalty as $18 \times np \ + 18 \times n(1 - p) = 18n$.

When the branch instruction is executed in a loop, the corresponding hint instruction and the inserted NOP instructions must also be executed within the loop so to take effect at each iteration. Therefore, the number of times the inserted NOP instructions executed will be the same as the number of times the branch instruction is executed. Moreover, each pair of *nop* and *lnop* instructions can be executed in one cycle, thanks to the dual-issue pipeline in the SPU. Therefore, the overhead for executing the inserted NOP instructions can be modeled as the follows:

$$Overhead_{pad} = n(n_{NOP} + 1)/2 \tag{25.6}$$

So far, we have discussed the branch penalties before and after NOP padding and the extra overhead for the execution of the inserted NOP instructions. The impact of NOP padding on performance can be now modeled as follows:

$$Profit_{pad} = Penalty_{no\_pad} - Penalty_{pad} - Overhead_{pad} \tag{25.7}$$

Clearly, the NOP padding should be carried out only when the calculated number is positive.

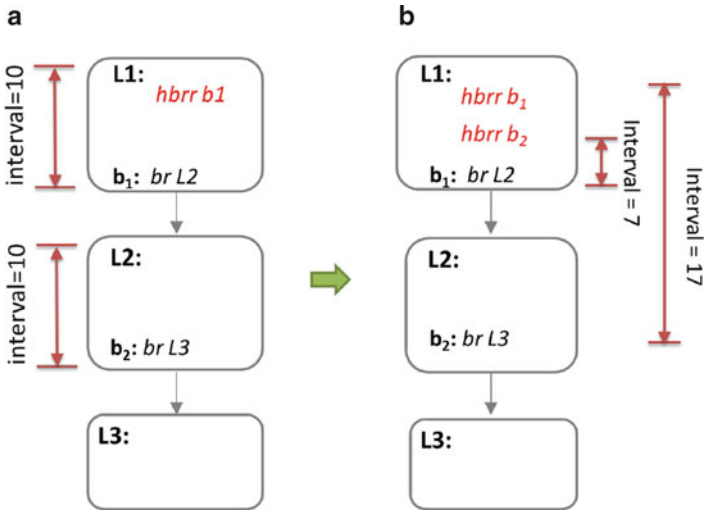


Fig. 25.6 Hint pipelining makes use of the minimum interval required to activate a hint instruction to insert extra hint instructions

Hint Pipelining. When two branches are close to each other, the hint instructions for the branches may interfere with each other. Figure 25.6a shows an example on how the SPU GCC compiler deals with such case. The SPU GCC compiler first tries to insert hints for both branches b_1 and b_2. However, when the hint instruction for b_2 has to be placed before b_1 to create a sufficiently large interval, it may overwrite the hint instruction for b_1. In this case, the GCC compiler estimates the priorities of the two branches, decides b_2 should be prioritized, and thus discards the hint for b_1. This problem is nevertheless not unsolvable, by prudently choosing the locations for both hints.

From the previous discussion, we have learned that a hint instruction will not be recognized by a branch instruction if it is executed within eight cycles before the branch. This gives us an opportunity to hint both branches. Figure 25.6b shows the method to hint two branches that are very close to each other. Although the hint instruction for b_2 is inserted before b_1, the interval between them is less than eight cycles. When b_1 is executed, the second hint is not recognized, so b_1 can be hinted correctly. However, when b_2 starts to execute later, the second hint will have been set up properly and take effect. With this approach, both b_1 and b_2 can be hinted. This method is called hint pipelining, as it "pipelines" hint instructions in the sense that the execution of the hint instructions are overlapped.

Again, to make sure this method is profitable, we need a cost model to find out the cost before applying this method and after applying this method. Assume in the given example in Fig. 25.6b that l_x denotes the number of instructions in the basic block L_x, p_x denotes the taken probability of the branch b_x, and n_x denotes the number of times b_x is executed. Keep in mind that we assume each instruction takes

one cycle, so l_x can be equally understood as the number of cycles that L_x takes to execute. The branch penalties before applying the hint pipelining method can be then calculated as follows:

$$\begin{aligned} Penalty_{no\_pipeline} = & \ Penalty(0, n_1, p_1) \\ & + (1 - p_1) \cdot Penalty(l_1 + l_2, n_2, p_2) \\ & + p_1 \cdot Penalty(0, n_2, p_2) \end{aligned}$$

The first term on the right-hand side is the branch penalty for b_1. Originally b_1 is not hinted, which can be viewed as if the interval is 0 cycle. The second and third term on the right-hand side are the branch penalties for b_2 when b_1 is not taken and when it is taken, respectively. When b_1 is not taken, b_2 will be hinted, and the interval between it and its hint is the sum of the number of instructions in both basic blocks L_1 and L_2; on the other hand, when b_1 is taken, b_2 will not be hinted, since the control flow will be diverted to a different basic block.

After hint pipelining is applied, both branches are hinted, although the interval between branch b_2 and its hint is decreased from $l_1 + l_2$ to $7 + l_2$ in the example. The branch penalties are changed as follows:

$$\begin{aligned} Penalty_{pipeline} = & \ Penalty(l_1, n_1, p_1) \\ & + (1 - p_1) \cdot Penalty(7 + l_2, n_2, p_2) \\ & + p_1 \cdot Penalty(0, n_2, p_2) \end{aligned}$$

Notice l_1 should be at least eight for this method to pan out, since otherwise the hint for b_1 will still not be recognizable.

The overhead of hint pipelining is the number of times the newly introduced hint instruction for b_1 is executed. Since the hint is inserted in basic block L_1, the number of times it is executed will be the same as b_1. The overhead is therefore as follows:

$$Overhead_{pipeline} = n_1 \tag{25.8}$$

The impact of hint pipelining on performance can be modeled as follows:

$$\begin{aligned} Profit_{pipeline} = & \ Penalty_{no\_pipeline} \\ & - Penalty_{pipeline} \\ & - Overhead_{pipeline} \end{aligned} \tag{25.9}$$

We should apply hint pipelining method only if the calculated number is positive.

Loop Restructuring. The NOP padding and hint pipelining methods are applicable when there is no loop or in the innermost loop. The loop restructuring

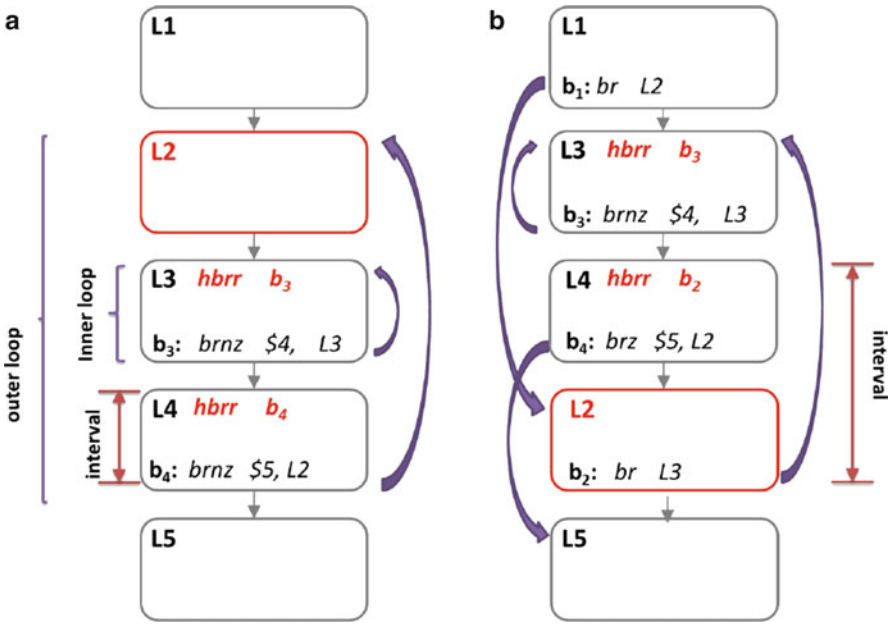


Fig. 25.7 Loop restructuring increases the leeway of hinting b_4 from l_4 in (**a**) to $l_2 + l_4$ in (**b**). Notice b_4 in (**a**) becomes b_2 in (**b**)

method, on the other hand, can be applied to outer loops in nested loops. This is done by altering the order of the basic blocks of the loops while keeping the semantic unchanged.

Figure 25.7 shows an example of nested loop restructuring method to reduce branch penalties in nested loops. Originally in Fig. 25.7a, b_4 is the condition of the outer loop, and the hint instruction for the branch b_4 is limited within the basic block L_4. This is because L_4 is preceded by a loop that consists of one basic block L_3. If the hint instruction for b_4 is inserted in any other basic block (earlier than L_4), it needs to either wait for the execution of all the iterations of L_3 (if the hint is inserted earlier than L_3), or it will be executed in every iteration of L_3 (if the hint is inserted in L_3). Neither case will lead to satisfactory performance. After restructuring as in Fig. 25.7b, L_2 is moved after L_4. To maintain the semantic, two unconditional branches from L_1 to L_2 and from L_2 to L_4 are introduced, respectively, and the condition of L_4 is modified accordingly. Such restructuring essentially turns b_2 into the condition of the outer loop. As a result, the possible location to insert a hint for b_2 is now increased to cover both L_4 and L_2.

Again, let l_x, p_x, and n_x, respectively, denote the number of instructions in the basic block L_x, the taken probability of the branch b_x, and the number of times b_x is executed. The branch penalties before restructuring of loops are as follows:

$$Penalty_{no\_reorder} = Penalty(l_3, n_3, p_3) + Penalty(l_4, n_4, p_4) \tag{25.10}$$

The branch penalties after restructuring the loops are as follows:

$$Penalty_{reorder} = 18 + Penalty(l_2 + l_4, n_2, p_2) + Penalty(l_3, n_3, p_3) + 18$$

Here the two 18s are the penalties for b_1 when entering the outer loop for the first time and for b_4 when exiting the outer loop when it is done, respectively.

The overhead of loop restructuring depends on the original size of basic block L_4. If it has less than eight instructions, then originally no hint can be made for b_4. After the restructuring, we will introduce a new hint in basic block L_4, which will be executed n_4 times. Otherwise, if originally there are more than eight instructions in L_4, no extra hint is introduced, and the overhead is zero. Therefore, the overhead for this method is as follows:

$$Overhead_{reorder} = \begin{cases} n_4, & if\ l_4 < 8 \\ 0, & otherwise \end{cases} \tag{25.11}$$

As a further optimization, some of the hint instructions can be promoted to be outside of a loop to avoid repeated computations. For example, Fig. 25.8a shows the code after promoting the hint for b_3 in the code given in Fig. 25.7a. After the promotion, the hint instruction only needs to be executed once while still

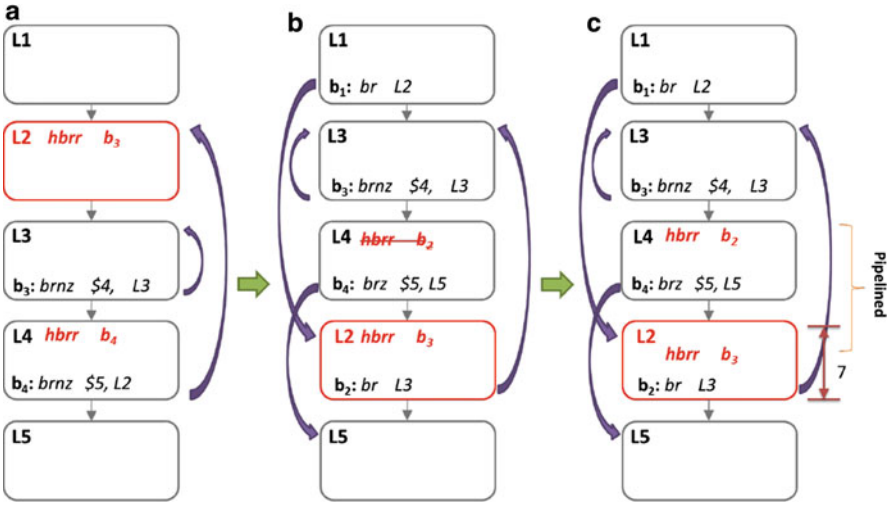


Fig. 25.8 The hint for b_3 is promoted to $L2$ from $L3$ in (**a**). This however may cause problems after loop restructuring as in (**b**), when the hint for b_3 overwrites the hint for b_2. Hint pipelining can be applied to enable both branches being hinted as in (**c**)

maintaining the semantics of the code. However, after the restructuring of nested loops as in Fig. 25.8b shows, the hint for b_3 overwrites the hint for b_2. This problem can be solved by applying the hint pipelining technique, as shown in Fig. 25.8c. Notice the promotion of a hint instruction should be applied only if the basic block the hint is promoted to itself does not have any taken branches, e.g., conditional or unconditional branches; otherwise, the promoted hint may interfere with the hints for the taken branches.

The three methods of reducing branch penalties – NOP padding, hint pipelining, and nested loop restructuring – can be combined and integrated into the compiler, as an optimization pass. The pass first restructures nested loops. It then traverses the Control-Flow Graph (CFG) of each function, in a bottom-up manner. That is to say, the pass first visits the bottom node (last basic block), and then recursively goes up along its predecessors. Once a branch is identified, the pass tries to promote its hint to its basic block whenever possible: if there is a branch that is likely taken in the predecessor, the traversal stops and the hint is inserted in the basic block the stop happens; otherwise, the pass keeps going up until it meets a basic block with any likely taken branch, or the basic block is the root of the CFG. Notice that a compiler with this pass enabled needs three extra parameters other than the input program, i.e., d, f, and s.

A microarchitecture-aware compiler is extremely important to improve the power saving and performance of execution in a software branch hinted processors. Figure 25.9 shows the performance improvement of the presented heuristic when being compared to the software branch hinting scheme provided by the SPU GCC compiler. The benchmarks that spend less than or equal to 20% of overall execution time for branches are considered with *low* branch penalty, while the others are considered as with *high* branch penalty. The heuristic outperforms the GCC scheme in every benchmark, and in general, the higher the ratio of branch penalty, the more the performance gain from applying the presented heuristic.

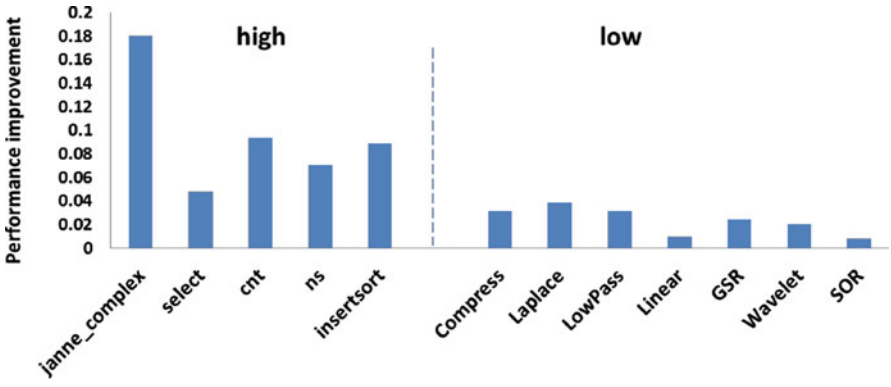


Fig. 25.9 Performance improvement of the presented heuristic is as much as 18% compared to the SPU GCC

25.1.2 Hardware-Aware Compilers for Design Space Exploration

Hardware-aware compilers are especially important for the design of embedded processors. At high level, the embedded processor design comprises of figuring out the microarchitectural configuration of the processor that will result in the best power and performance characteristics. Traditional DSE relies solely on simulation, as shown in Fig. 25.10. The same (compiled) code is measured on architectural models with different design parameters. The design parameters that yield the most desirable outcome are chosen. However, using the same code for different architectural variations may not guarantee fairness of the comparison, since the optimal code generated may vary as the design parameters changes. For example, loop tiling divides the iteration space into tiles or blocks to better fit the data cache. If we change the cache parameters, such as cache-line size or cache associativity, then we may need to change the size of each tile. Therefore, to be able to accurately explore the design space, a hardware-aware compiler should be included in the loop of DSE, so that every time the architectural design parameters are changed, the code generation should be adjusted accordingly, by compiling with the changed design parameters. Figure 25.11 shows the example of a framework of CIL DSE. CIL DSE is especially important in embedded systems, where the hardware-aware compiler can have a very notable impact on the power and performance characteristics of the processor.

In the rest of this subsection, we will study PBExplore – a framework for CIL DSE of partial bypassing in embedded processors. At the heart of the

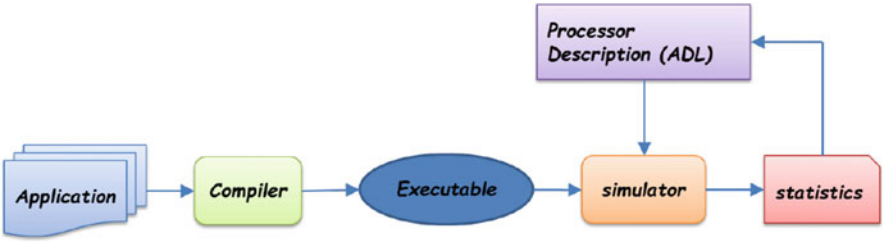


Fig. 25.10 Traditional DSE relies solely on simulation

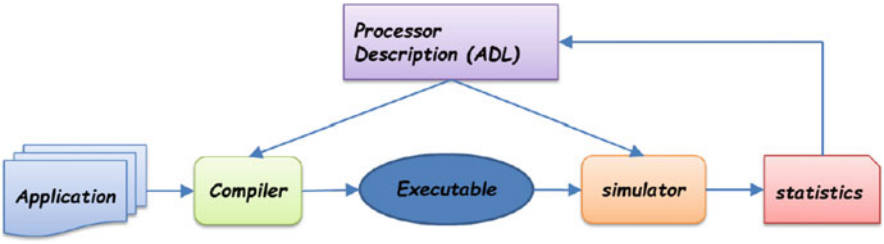


Fig. 25.11 CIL DSE includes the compiler in the loop of exploring best design parameters

PBExplore is a compiler that can generate high-quality code for a given partial bypassing configuration. Such a compiler can be used to explore different bypass configurations and discover the one that offers the best power, performance, cost, and complexity trade-offs.

25.1.2.1 The Case for Partial Bypassing

Pipelining is a widely used technique in modern processors to explore instruction-level parallelism and allow processors to achieve much higher throughput. However, the presence of hazards in the pipeline greatly impairs its value as they stall the pipeline and cause significant performance loss. Consequently, techniques are proposed to resolve the problems [10, 23]. Bypassing, also known as operand forwarding, is a popular solution to reduce data hazards. Bypassing adds additional datapaths and control logic to the processor so that the result of an operation can be forwarded to subsequent dependent operations even before it is written back to the register file.

While bringing in the great benefit, bypasses increase design complexity and may introduce significant overhead. Bypasses are often included in time-critical datapaths and therefore cause pressure on cycle time, especially the single cycle paths. This is particularly important in wide issue machines, where the delay may become more significant – due to extensive bypassing very wide multiplexors or buses with several drivers may be needed. Partial bypassing presents a trade-off between the performance, power, and cost of a processor and is therefore an especially valuable technique for application-specific embedded processors. Also, note that adding or removing bypasses or the bypass configuration of the processor does not affect its ISA.

25.1.2.2 Operation Latency-Based Schedulers Cannot Accurately Model Partial Bypassing

Traditionally, the retargetable compiler uses constant operation latency of each operation to detect and avoid data hazards [23]. The *operation latency* of an operation o is defined as a positive integer $ol \in I+$, such that if any data-dependent operation is issued more than ol cycles after issuing o, there will be no data hazards. When no bypassing or complete bypassing (the result of an operation can be forwarded at every stage of a pipeline once it is calculated and before it is committed to the memory system) is implemented in a pipeline, the operation latency is constant, and therefore the retargetable compiler can work perfectly. However, the presence of partial bypassing (the result of an operation can be forwarded only at some stage(s) but not all the stages of a pipeline) introduces variable operation latency and poses challenges for such a compiler.

To better understand the challenge of designing retargetable compilers in the presence of partial bypassing, let us first consider the differences of pipelines without bypassing, with complete bypassing, and with partial bypassing. Figure 25.12 illustrates the execution of an ADD operation in a simple five-stage pipeline without any bypassing. In the absence of any hazards, if the ADD operation is in F pipeline stage in cycle i, then it will be in OR pipeline stage in cycle i + 2. At this time, it

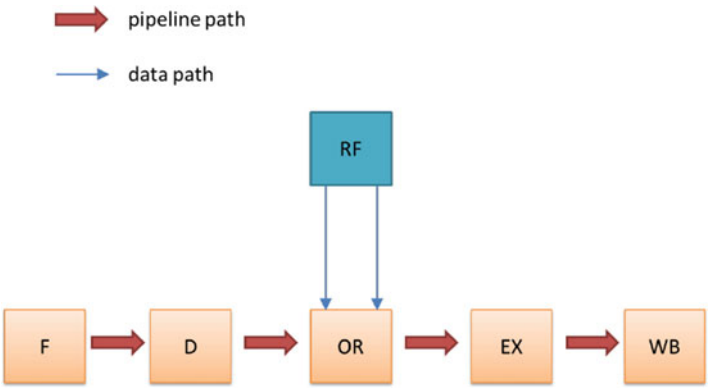


Fig. 25.12 A 5-stage processor pipeline with no bypassing

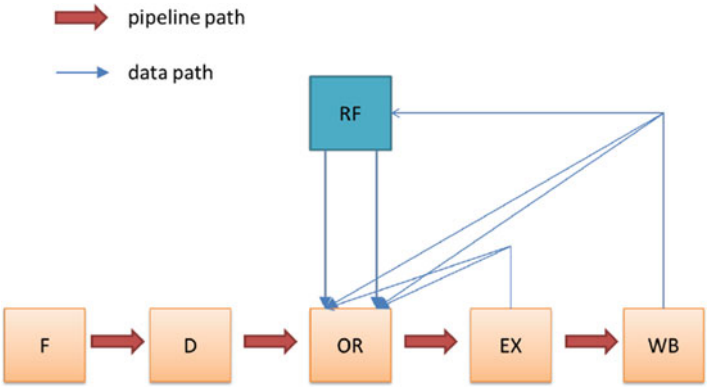


Fig. 25.13 A 5-stage processor pipeline with complete bypassing

reads the two source registers. The ADD operation then writes back the destination register in cycle i + 4, when it reaches the WB pipeline stage. The result of the ADD operation can be read from the register file in and after cycle i + 5. The operation latency of the ADD operation is three cycles $((i+5) - (i+2))$, so any instructions that are dependent on the result of the current instruction have to be scheduled at least three cycles later to avoid the data hazards. Figure 25.13 shows the pipeline with complete bypassing. The pipeline now includes forwarding paths from both execution (EX) and write back (WB) stages to both the operands of operand reading (OR) stage. The operation latency now becomes one cycle, since any dependent instructions scheduled one or two cycles after the ADD instruction can read its result from the bypasses, while those scheduled three or more cycles later can read the result from RF. Finally, we show an example of partial bypassing in Fig. 25.14. The pipeline only contains bypasses from EX (but not WB) stage to both the operands of OR pipeline stage. In this circumstance, scheduling a data-dependent operation one

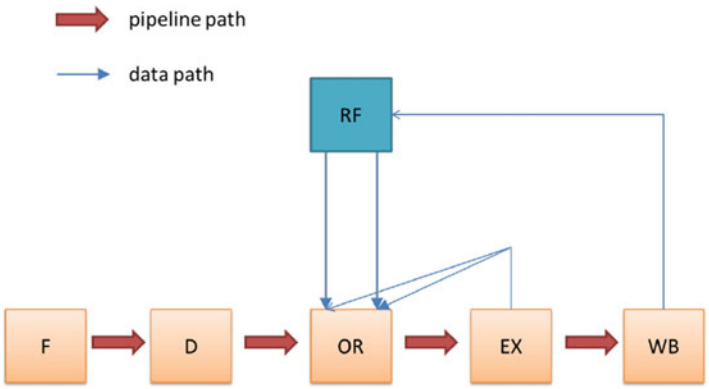


Fig. 25.14 A 5-stage processor pipeline with partial bypassing

or three cycles after the ADD will not result in a data hazard, since its result can be read either from EX pipeline stage via the bypasses or from the register file after the ADD operation writes back its result. However, if the data-dependent operation is scheduled two cycles after the scheduling ADD operation which is currently in WB stage, then it cannot do anything but wait, since no bypasses from WB are present. A data hazard happens. The operation latency of ADD in the partial bypassed pipeline in Fig. 25.14 is denoted by one, three, which means that scheduling a data-dependent operation one or three or more cycles after the schedule cycle of ADD will not cause any data hazard, but scheduling the data-dependent operation two cycles after the schedule cycle of ADD will cause a data hazard. The operation latency becomes nonconstant under partial bypassing. As a result, partial bypassing paralyzes the traditional retargetable compilers, which assumes a constant operation latency to detect pipeline hazards.

Without the accurate pipeline hazard detection technique, the retargetable compiler has to either conservatively assume no bypassing is present or aggressively assume that the pipeline is completely bypassed. However, both approaches will result in suboptimal code generated. To solve this problem, Operation Table (OT) [28] can be employed. An OT maintains the snapshot of the processor resources an operation uses in each cycle of its execution. It takes into consideration the (partial) bypassing in the pipeline and can therefore detect data hazards in advance even for partially bypassed processors. Besides, as the OT records at each cycle which processor resources are used, it is able to detect the structural hazards as well. As a result, an OT-based scheduler can accurately detect and avoid pipeline hazards and improve processor performance.

25.1.2.3 OT to Accurately Model the Execution of Operations in a Pipeline

In this subsection, we present the concept of OT that can accurately model the execution of operations in a processor pipeline, and later we will use them to

Table 25.2 Definition of the OT

OperationTable	:=	{ otCycle }
otCycle	:=	unit ros wos bos dos
ros	:=	ReadOperands { operand }
wos	:=	WriteOperands { operand }
bos	:=	BypassOperands { operand }
dos	:=	DestOperands { regNo }
operand	:=	regNo { path }
path	:=	port regConn port regFile

Table 25.3 The OT of ADD R1 R2 R3

1	F	
2	D	
3	OR	
	ReadOperands	
	R2	
		p1, C1, p6, RF
	R3	
		p2, C2, p7, RF
		p2, C5, p3, EX
	DestOperands	
	R1, RF	
4	EX	
	BypassOperands	
	R1	
		p3, C5, p2, OR
5	WB	
	WriteOperands	
	R1	
		p4, C3, p8, RF

develop a bypass-aware instruction scheduler. An OT describes the execution of one operation in the processor. Table 25.2 shows the grammar of an OT. Each entry, *otCycle*, in an OT describes the state of the operation in that execution cycle in the pipeline. *otCycles* are sorted in temporal order. Each *otCycle* records in that cycle the pipeline unit the operation is in (*unit*), the operands it needs to read (*ros*), write (*wos*), or bypass (*bos*), and the destination registers (*dos*) the operation may write to. Each operand (*operand*) is defined by the register number (*regNo*) and all the possible paths it may be transferred. Each possible path (*path*) consists of the ports (*port*), register connections (*regConn*), and the register file (*regFile*).

Table 25.3 shows the OT of executing an add operation, *ADD R1 R2 R3*, on the partially bypassed pipeline shown in Fig. 25.15. Without loss of the generality, assume by the time the add operation starts to execute, no hazards are present, so the add operation can be executed in five cycles, and consequently the OT of this

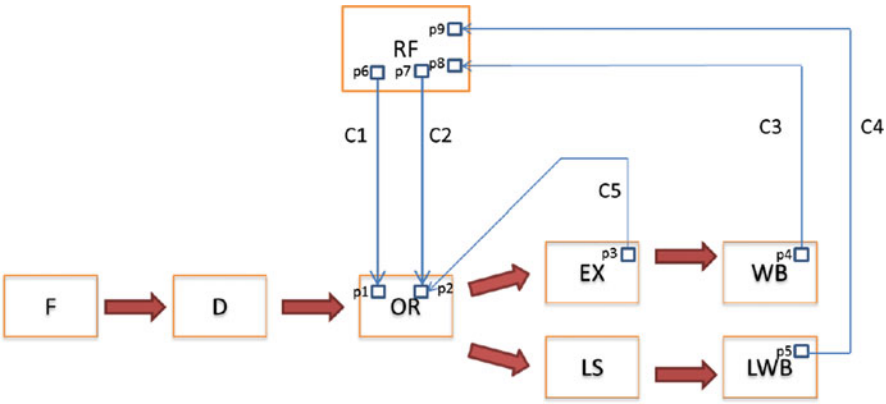


Fig. 25.15 An example partially bypassed pipeline

add operation contains five *otCycles*. The add operation is fetched to the *F* pipeline unit and decoded at *D* pipeline unit, respectively, during the first two cycles. In the third cycle, the add operation proceeds to *OR* pipeline unit and reads its source operands *R2* and *R3*. All the possible paths to read the each operand are included in the table. The first operand *R2* can be read only from the register file *RF* via connection *C1*, while the second operand *R3* can be read either from the register file *RF* via the connection *C2* from port *p7* to *p2* or from the pipeline unit *EX* via connection *C5* from port *p3* to port *p2*. In addition to the source operands, the destination operands *R1* are listed as well, and the dependent operations should be scheduled after accordingly. In the fourth cycle, the add operation is sent to the pipeline unit *EX* for execution. At the end of this cycle, the result of the operation is calculated and available for bypassing. The operation at the *OR* unit at that time can read the calculated result as its second operand via connection *C5*. In the last cycle, the operation is written back to *R1* from *WB* pipeline unit to the register file *RF* via connection *C3*.

There may be multiple paths to read each operand in the presence of bypasses. As an example, the Intel XScale processor provides seven possible bypasses for each operand, in addition to the register file. The OT of an operation lists all the possible paths to read each operand. As a result, the OT may potentially have to store eight paths (seven from bypasses plus one from the register file) for each an operand to read. To prevent such superfluity to consume too much space in the OT, the concept of Bypass Register File (BRF) is introduced. A BRF is essentially a virtual register file that serves as a temporary storage for each operand having bypasses. All the values bypassed to the operand are first written to the BRF and then read by the operation. A value from the bypass must be read in the same cycle once the value is calculated; therefore, each value can exist only for one cycle in the BRF. Each operand needs only one BRF to accept values from all the bypasses that attach to it. As a result, the space consumed by an OT can be greatly reduced.

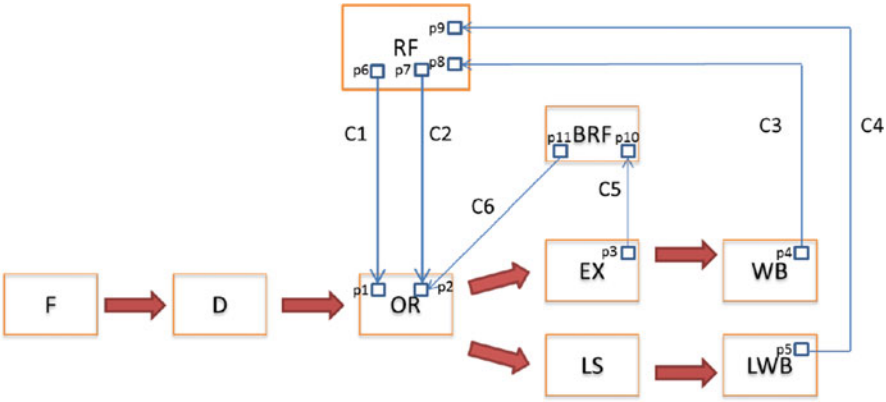


Fig. 25.16 An example partially bypassed pipeline with BRF

Table 25.4 The OT of ADD R1 R2 R3 with BRF

1	F	
2	D	
3	OR	
	ReadOperands	
	R2	
		p1, C1, p6, RF
	R3	
		p2, C2, p7, RF
		p2, C6, p11, BRF
	DestOperands	
	R1, RF	
4	EX	
	BypassOperands	
	R1	
		p3, C5, p10, BRF
5	WB	
	WriteOperands	
	R1	
		p4, C3, p8, RF

Figure 25.16 shows the processor pipeline with a BRF. The bypassed result from the *EX* unit to the second source operand of the *OR* unit is first written to the BRF via connection *C5*. The *OR* unit then reads the value either from the BRF via connection *C6* or from actual register file *RF* via connection *C2*. Table 25.4 shows the OT of the operation *ADD R1 R2 R3* in the pipeline with the BRF in Fig. 25.16. The only differences are that the second source operand of *OR* unit can be read from either the actual register file *RF* or the virtual register file *BRF*, and the result of the *EX* unit is now first bypassed to the *BRF* instead of directly to the second source operand of the *OR* unit.

Detecting Pipeline Hazards Using OT

To illustrate the power OT possesses in detecting pipelining hazards, let us consider an example of applying OT-based scheduling in the pipeline in Fig. 25.16 on the three operations as follows:

MUL R1 R2 R3 ($R1 \leftarrow R2 \times R3$)
ADD R4 R2 R3 ($R4 \leftarrow R2 + R3$)
SUB R5 R4 R2 ($R5 \leftarrow R4 - R2$)

Assume both SUB and ADD operations take one cycle in the *EX* stage and the MUL operation spends two cycles in the same stage. In addition, all the pipeline resources are available initially. Therefore, when the MUL operation is scheduled at the first cycle, there will be no hazards. Table 25.5 shows the state of the machine after MUL is scheduled.

We then try to schedule ADD in the next cycle. However, a resource hazard will happen, since the *EX* pipeline unit is still busy executing the second cycle of *MUL* operation. Table 25.6 shows the state of the processor pipeline after scheduling ADD in the second cycle. A resource hazard is detected when the fourth *otCycle* of ADD is tried in the fifth cycle, so the *otCycle* should be not be scheduled at this cycle.

At this point, if we keep scheduling the SUB operation in the third cycle, a data hazard will be detected. The SUB operation needs to read the value of the first operand *R4*, which is calculated by the previous ADD operation. However, the

Table 25.5 Pipeline states after scheduling MUL R1 R2 R3 in Cycle 1

Cycle	Busy resources	!RF	BRF
	Operation1		
1.	F	–	–
2.	D	–	–
3.	OR, p1, C1, p6, p2, C2, p7	–	–
4.	EX	R1	–
5.	EX, p3, C4, p10	R1	R1
6.	WB, p4, C3, p8	R1	–
7.		–	–

Table 25.6 Pipeline states after scheduling ADD R4 R2 R3 in Cycle 2

Cycle	Busy resources		!RF	BRF
	Operation1	Operation 2		
1.	F		–	–
2.	D	F	–	–
3.	OR, p1, C1, p6, p2, C2, p7	D	–	–
4.	EX	OR, p1, C1, p6, p2, C2, p7	R1	–
5.	EX, p3, C4, p10	Resource hazard	R1 R4	R1
6.	WB, p4, C3, p8	EX, p3, C4, p10	R1 R4	R4
7.		WB, p4, C3, p8	R4	–
8.			–	–

Table 25.7 Pipeline states after scheduling SUB R5 R4 R2 in cycle 3

Cycle	Busy resources			!RF	BRF
	Operation1	Operation 2	Operation 3		
1.	F			–	–
2.	D	F		–	–
3.	OR, p1, C1, p6, p2, C2, p7	D	F	–	–
4.	EX	OR, p1, C1, p6, p2, C2, p7	D	R1	–
5.	EX, p3, C4, p10	Resource hazard	Data hazard	R1 R4	R1
6.	WB, p4, C3, p8	EX, p3, C4, p10	Data hazard	R1 R4	R4
7.		WB, p4, C3, p8	Data hazard	R4	–
8.			OR, p1, C1, p6, p2, C2, p7	R5	–
9.			EX, p3, C4, p10	R5	R5
10.			WB, p4, C3, p8	R5	–
11.				–	–

bypass in Table 25.5 is from *EX* pipeline unit to the second operand in *OR* pipeline unit, so there will not be any available path for this operand to be transferred at the time the SUB operation enters to the *EX* pipeline unit. The data hazard is resolved in the eighth cycle. Table 25.7 shows that the state of the processor pipeline after SUB is scheduled in the third cycle. After the scheduling of the SUB operation, all the operations are successfully scheduled with both data and resource hazards detected, even in the presence of partial bypassing.

25.1.2.4 List Scheduling Algorithm Using OT

In the presence of partial bypassing, the operation latency of an operation is not sufficient to avoid all the data hazards – OT is needed. The traditional list scheduling algorithm can be very easily modified by using OT. Figure 25.17 shows the list scheduling algorithm that uses OT for pipeline hazard detection. The *DetectHazard* function (line 10) and the *AddOperation* function (line 13) are two functions that are based on OT. The *DetectHazard* function checks if scheduling all *otCycles* of the operation v starting from the machine cycle t will cause any hazards. Once the scheduler finds the earliest available machine cycle, it calls *AddOperation* function to schedule operation v in cycle t.

Experiments are performed in the Intel XScale microarchitecture to verify the capability of the OT-based scheme. Figure 25.18 shows the pipeline of XScale architecture. The experimental setup is shown in Fig. 25.19. Each benchmark application is first compiled with the GCC cross compiler for Intel XScale processor. The OT-based scheduling is then applied to each basic block of the original program to generate the executable again. The two versions of executable files are then run on the XScale cycle-accurate simulator, respectively. Performance is measured as the number of cycles spent on executing applications. The improvement of performance is measured as $(gccCycles - otCycles) * 100/gccCycles$, where $gccCycles$ is

Fig. 25.17 List scheduling algorithm using OT

ListScheduleUsingOTs(V)
01: $U = V - v_0; F = \varphi; S = v_0$

/* initialize */
02: **foreach** $(v \in V)$
03: $schedTime[v] = 0$
04: **endFor**

/* list schedule */
05: **while** $(U \neq \varphi)$
06: $F = \{v | v \in U, parents(v) \subset S\}$
07: $F.sort()$ /* some priority function */
08: $v = F.pop()$
09: $t = MAX(schedTime(p)),\ p \in parents(v)$
10: **while** $(DetectHazard(machineState, v.OT, t))$
11: $t++$
12: **endWhile**
13: $AddOperation(machineState, v.OT, t)$
14: $schedTime[v] = t$
15: **endWhile**

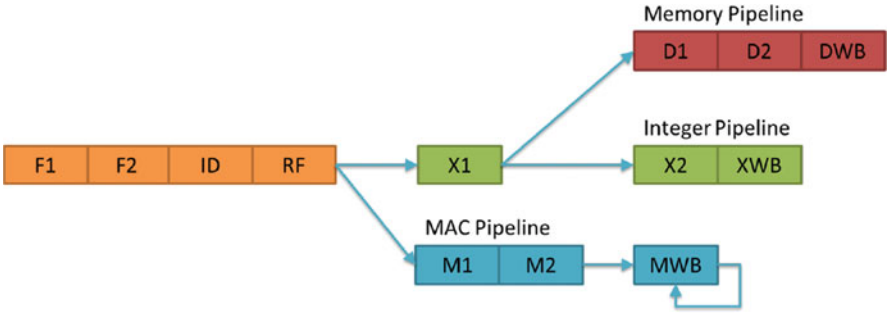
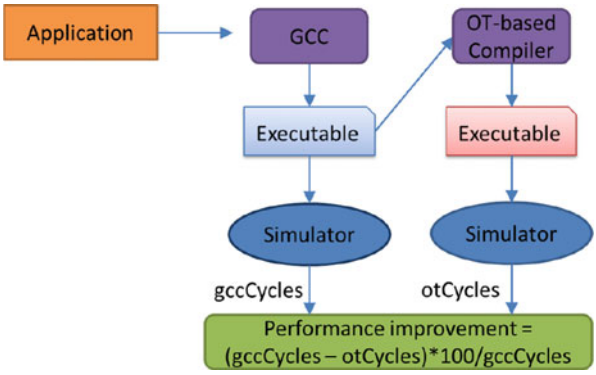


Fig. 25.18 The pipeline in XScale

Fig. 25.19 Experimental setup



the number of cycles spent on running an applications compiled with GCC compiler, and $otCycle$ is its counterpart with OT-based scheduling. Figure 25.20 shows the details. The OT-based scheme improves the performance over the GCC compiler by up to 20%.

25.1.2.5 CIL Partial Bypass Exploration

With effectiveness of OT-based scheme, we can further make use of it to explore the design space of partial bypassed processors. PBExplore, a CIL Framework for DSE of partial bypassing in processors is proposed to accommodate the need. The compiler in the PBExplore takes as input bypass configuration, as shown in Fig. 25.21. A bypass configuration describes the pipeline stage each individual bypass starts (source), and the operand that can consume the value the bypass transfers (destination). The binary generated by the bypass-aware compiler is

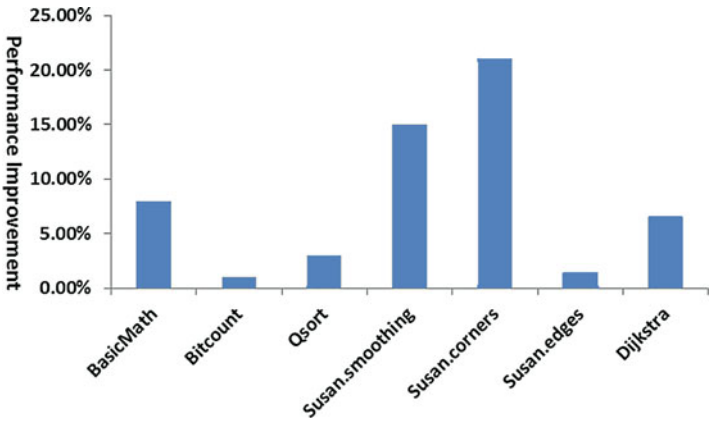


Fig. 25.20 Performance improvement of the compiler with OT-based scheduling over the GCC compiler

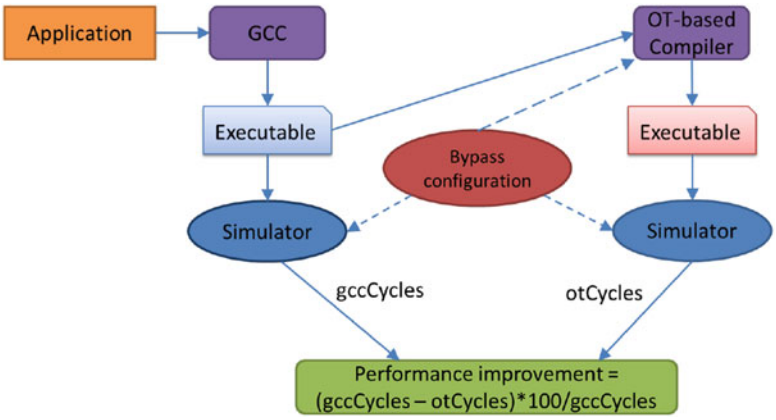


Fig. 25.21 PBExplore: A CIL framework for partial bypass exploration

then run on a cycle-accurate simulator that is parameterized on the same bypass configuration. The simulator then dumps the estimations of cycles of execution, area, and power consumption.

PBExplore can effectively guide designers to use the best design decisions and avoid suboptimal design decisions that may happen in simulation-only DSE. Figures 25.22, 25.23 and 25.24 show, respectively, the change of execution cycles when only the X-bypasses (enabling bypassing of the pipeline stages in the integer pipeline in Fig. 25.18), D-bypasses (enabling bypassing of the pipeline stages in the memory pipeline), and M-bypasses (enabling bypassing of the pipeline stages in the Multiply-Accumulator (MAC) pipeline) are varied, respectively, while the other two bypasses are fixed. Figure 25.22 shows that while the execution cycles for configurations < X2 X1 >, < XWB X1 > and < XWB X2 > are similar under

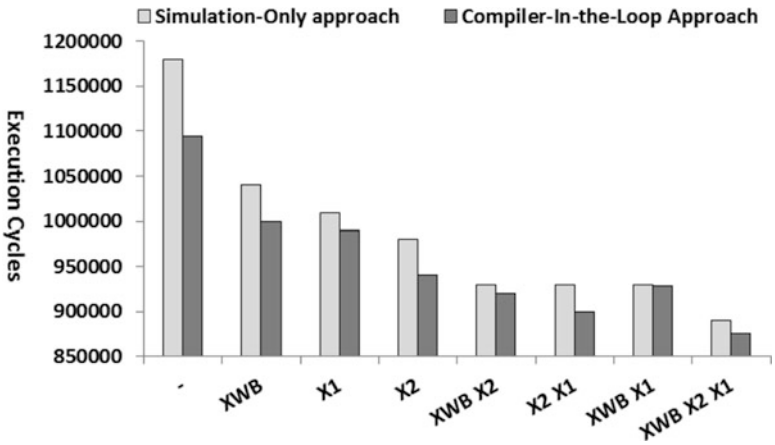


Fig. 25.22 X-bypass exploration for the bitcount benchmark

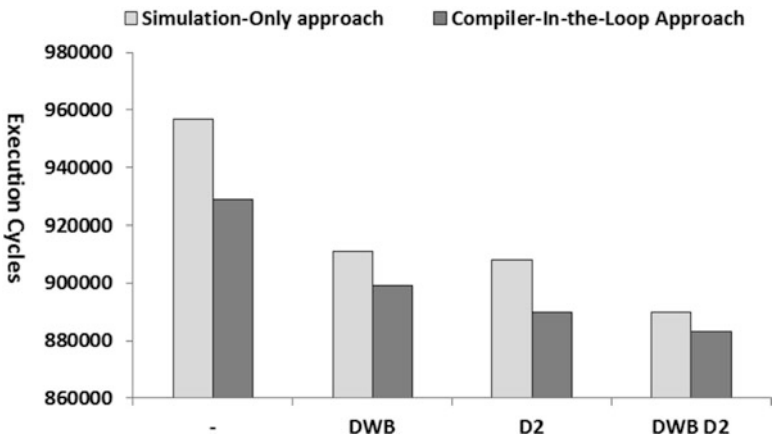


Fig. 25.23 D-bypass exploration for the bitcount benchmark

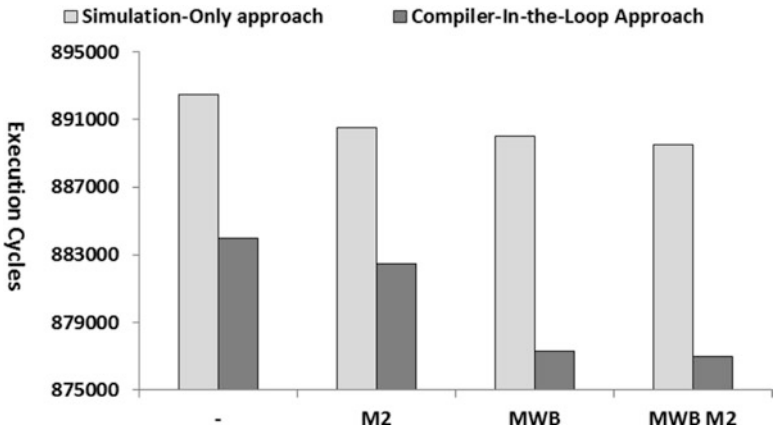


Fig. 25.24 M-bypass exploration for the bitcount benchmark

simulation-only approach, the bypass-sensitive compiler is able to identify < X2 X1 > the best among the three choices. If designers choose to have only two bypasses in the processor, then they would have made the wrong choice based on simulation solely. Similarly, Fig. 25.23 shows that when there is only one bypass, bypassing *D2* pipeline stage is a better choice than bypassing *DWB* according to PBExplore, while the simulation-only approach may mislead designers to bypass either *D2* or *DWB*. Similar observations can be found for the M-bypass exploration in Fig. 25.24 and the D-bypass exploration in Fig. 25.23.

25.1.3 Conclusions

Embedded systems are ubiquitous in our daily life, ranging from portable music players to real-time control systems in space shuttles. The diversity of embedded applications eventually boils down to multidimensional design constraints on embedded systems. To meet these constraints, embedded processors often feature unique design parameters, several missing features, and often quite quirky designs. For these embedded processors, the compiler often has a very significant impact on the power and performance characteristics of the processor – and therefore hardware-aware compilers are most useful and effective for embedded processors. Hardware-aware compilers take the microarchitectural description of the processor into account in addition to the application code in order to compile. There are two main use-cases for hardware-aware compilers. The first one is the traditional use, i.e., as a production compiler for an embedded processor. In addition to this, a hardware-aware compiler can be used to design an efficient embedded processor. The hardware-aware compiler enables the CIL DSE of the microarchitectural space of the processor, which takes into consideration the effects compilers have on the power consumption and performance of the processor. We demonstrate these two

uses by presenting a compiler technique to significantly alleviate branch penalties in processors without hardware branch prediction in Sect. 25.1.1, and a OT-based compiler technique that can be used to improve the performance and to help the design of processors with partial bypassing in Sect. 25.1.2. The experimental results corroborate the importance of hardware-aware compilation.

References

1. Bala V, Rubin N (1995) Efficient instruction scheduling using finite state automata. In: Proceedings of the 28th annual international symposium on microarchitecture, pp 46–56. doi:10.1109/MICRO.1995.476812
2. Ball T, Larus JR (1993) Branch prediction for free. In: Proceedings of PLDI. ACM, New York, pp 300–313. doi:10.1145/155090.155119
3. Chen T, Raghavan R, Dale JN, Iwata E (2007) Cell broadband engine architecture and its first implementation – a performance view. IBM J Res Dev 51(5):559–572. doi:10.1147/rd.515.0559
4. Dual-Core Intel Itanium Processor 9000 and 9100 Series (2007). http://download.intel.com/design/itanium/downloads/314054.pdf
5. Flachs et al B (2006) The microarchitecture of the synergistic processor for a cell processor. IEEE Solid-State Circuits 41(1):63–70
6. Fog A (2008) The microarchitecture of Intel and AMD CPUs
7. GNU Toolchain 4.1.1 and GDB for the Cell BE's PPU/SPU. http://www.bsc.es/plantillaH.php?cat_id=304
8. Grun P, Dutt N, Nicolau A Memory aware compilation through accurate timing extraction. In: Proceedings of the 37th annual design automation conference, DAC'00. ACM, New York, pp 316–321 (2000). doi:10.1145/337292.337428
9. Grun P, Dutt N, Nicolau A (2000) MIST: an algorithm for memory miss traffic management. In: IEEE/ACM international conference on computer aided design, ICCAD-2000, pp 431–437. doi:10.1109/ICCAD.2000.896510
10. Grun P, Halambi A, Dutt N, Nicolau A (2003) RTGEN-an algorithm for automatic generation of reservation tables from architectural descriptions. IEEE Trans Very Large Scale Integr (VLSI) Syst 11(4):731–737. doi:10.1109/TVLSI.2003.813011
11. Halambi A, Grun P, Ganesh V, Khare A, Dutt N, Nicolau A (1999) EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In: Design, automation and test in Europe conference and exhibition 1999. Proceedings, pp 485–490. doi:10.1109/DATE.1999.761170
12. Hoffmann A, Schliebusch O, Nohl A, Braun G, Wahlen O, Meyr H (2001) A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA. In: Proceedings of the 2001 IEEE/ACM international conference on computer-aided design, ICCAD'01. IEEE Press, Piscataway, pp 625–630
13. https://gcc.gnu.org/ (2007)
14. IBM: Cell Broadband Engine Programming Handbook including PowerXCell 8i. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7A77CCDF14FE70D5852575CA0074E8ED
15. Intel Corporation. Intel XScale(R) Core: Developer's Manual. http://www.intel.com/design/iio/manuals/273411.htm
16. Keutzer K, Malik S, Newton A (2002) From ASIC to ASIP: the next design discontinuity. In: IEEE international conference on computer design: VLSI in computers and processors, 2002. Proceedings, pp 84–90. doi:10.1109/ICCD.2002.1106752
17. Kondo M, Kobyashi H, Sakamoto R, Wada M, Tsukamoto J, Namiki M, Wang W, Amano H, Matsunaga K, Kudo M, Usami K, Komoda T, Nakamura H (2014) Design and evaluation of

fine-grained power-gating for embedded microprocessors. In: Design, automation and test in Europe conference and exhibition (DATE), pp 1–6. doi:10.7873/DATE.2014.158
18. Kongetira P, Aingaran K, Olukotun K (2005) Niagara: a 32-way multithreaded sparc processor. IEEE Micro 25(2):21–29. doi:10.1109/MM.2005.35
19. Lattner C (2002) LLVM: an infrastructure for multi-stage optimization. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana. See http://llvm.cs.uiuc.edu
20. Leupers R (2000) Code generation for embedded processors. In: The 13th international symposium on system synthesis, 2000. Proceedings, pp 173–178. doi:10.1109/ISSS.2000.874046
21. Lowney PG, Freudenberger SM, Karzes TJ, Lichtenstein WD, Nix RP, O'Donnell JS, Ruttenberg JC (1993) The multiflow trace scheduling compiler. J Supercomput 7:51–142
22. Lu J, Kim Y, Shrivastava A, Huang C (2011) Branch penalty reduction on IBM cell SPUs via software branch hinting. In: Proceedings of CODES+ISSS, pp 355–364
23. Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco
24. Park D, Lee J, Kim NS, Kim T (2010) Optimal algorithm for profile-based power gating: a compiler technique for reducing leakage on execution units in microprocessors. In: 2010 IEEE/ACM international conference on computer-aided design (ICCAD), pp 361–364. doi:10.1109/ICCAD.2010.5653652
25. Patterson D, Anderson T, Cardwell N, Fromm R, Keeton K, Kozyrakis C, Thomas R, Yelick K (1997) A case for intelligent RAM. IEEE Micro 17(2):34–44. doi:10.1109/40.592312
26. Proebsting TA, Fraser CW (1994) Detecting pipeline structural hazards quickly. In: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'94. ACM, New York, pp 280–286. doi:10.1145/174675.177904
27. Roy S, Katkoori S, Ranganathan N (2007) A compiler based leakage reduction technique by power-gating functional units in embedded microprocessors. In: 20th international conference on VLSI Design, 2007. Held jointly with 6th international conference on embedded systems, pp 215–220. doi:10.1109/VLSID.2007.10
28. Shrivastava A (2006) Compiler-in-loop exploration of programmable embedded systems. Ph.D. thesis, Donald Bren School of Information and Computer Sciences
29. Shrivastava A, Issenin I, Dutt N (2005) Compilation techniques for energy reduction in horizontally partitioned cache architectures. In: Proceedings of the 2005 international conference on compilers, architectures and synthesis for embedded systems, CASES'05. ACM, New York, pp 90–96. doi:10.1145/1086297.1086310
30. Siska C (1998) A processor desription language supporting retargetable multi-pipeline DSP program development tools. In: Proceedings of the 11th international symposium on system synthesis, ISSS'98. IEEE Computer Society, Washington, DC, pp 31–36
31. Trimaran. http://www.trimaran.org/
32. Wagner TA, Maverick V, Graham SL, Harrison MA (1994) Accurate static estimators for program optimization. In: Proceedings of the ACM SIGPLAN 1994 conference on programming language design and implementation, PLDI'94. ACM, New York, pp 85–96. doi:10.1145/178243.178251
33. Wu Y, Larus JR (1994) Static branch frequency and program profile analysis. In: Proceedings of the 27th annual international symposium on Microarchitecture. ACM, New York, pp 1–11. doi:10.1145/192724.192725
34. Zivojnovic V, Pees S, Meyr H (1996) LISA-machine description language and generic machine model for HW/SW co-design. In: Workshop on VLSI signal processing, IX, pp 127–136. doi:10.1109/VLSISP.1996.558311

Memory-Aware Optimization of Embedded Software for Multiple Objectives

26

Peter Marwedel, Heiko Falk, and Olaf Neugebauer

Abstract

Information processing in Cyber-Physical Systems (CPSs) has to respect a variety of constraints and objectives such as response and execution time, energy consumption, Quality of Service (QoS), size, and cost. Due to the large impact of the size of memories on their energy consumption and access times, an exploitation of memory characteristics offers a large potential for optimizations. In this chapter, we will describe optimization approaches proposed by our research groups. We will start with optimizations for single objectives, such as energy consumption and execution time. As a consequence of considering hard real-time systems, special attention is on the minimization of the Worst-Case Execution Time (WCET) within compilers. Three WCET reduction techniques are analyzed: exploitation of scratchpads, instruction cache locking, and cache partitioning for multitask systems. The last section presents an approach for considering trade-offs between multiple objectives in the design of a cyber-physical sensor system for the detection of bio-viruses.

Acronyms

CFG	Control-Flow Graph
CPS	Cyber-Physical System
CPU	Central Processing Unit
CRPD	Cache-Related Preemption Delay
DRAM	Dynamic Random-Access Memory

P. Marwedel (✉) • O. Neugebauer
Computer Science, TU Dortmund University, Dortmund, Germany
e-mail: Peter.Marwedel@tu-dortmund.de; Olaf.Neugebauer@tu-dortmund.de

H. Falk
Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany
e-mail: Heiko.Falk@tuhh.de

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_27

FIFO	First-In First-Out
GA	Genetic Algorithm
GPU	Graphics Processing Unit
ILP	Integer Linear Program
LRU	Least-Recently Used
MMU	Memory Management Unit
PAMONO	Plasmon-Assisted Microscopy of Nano-Objects
QoS	Quality of Service
SPM	Scratchpad Memory
SRAM	Static Random-Access Memory
SVM	Support Vector Machine
WCC	WCET-aware C Compiler
WCEC	Worst-Case Energy Consumption
WCEP	Worst-Case Execution Path
WCET	Worst-Case Execution Time

Contents

26.1 Introduction

This chapter considers the mapping of software applications to execution platforms for embedded systems. Embedded systems are information processing systems embedded into enclosing products such as cars or smart homes [29]. In combination with their physical environment, embedded systems form so-called Cyber-Physical Systems (CPSs). According to the National Science Foundation (NSF), "*CPSs are engineered systems that are built from and depend upon the synergy of computational and physical components*" [31]. In our view, embedded systems can be seen as the information processing part in a CPS. Due to the integration with the physical environment, embedded systems have to meet a large set of functional requirements, constraints, and objectives. Hence, in addition to meeting the functional requirements, optimization for the relevant objectives within the design space imposed by the constraints is an essential part of design methodologies for embedded systems. Analyzing currently available technology, it turns out that much of the potential for optimizations concerns memories and their usage. In the following sections, the existence of this potential will be proved by means of examples. The examples are intended to provide an overview over optimization potential in this area, using our research results for demonstration. Specific pointers to our publications are included for further reference and more in-depth discussion.

26.2 Constraints and Objectives

One of the characteristics of embedded systems is the need to consider a large variety of constraints and objectives during their design.

26.2.1 Timing

Embedded systems often have to meet real-time constraints that make them real-time systems. Not completing computations within a given time can result in a serious loss of the quality provided by the system (e.g., if the audio or video quality is affected) or may cause harm to the user (e.g., if cars, trains, or planes do not operate in the predicted way). Time constraints are called *hard* if not meeting them could result in a catastrophe. All other time constraints are called *soft*.

During the design of real-time systems, the Worst-Case Execution Time (WCET) plays an important role. The WCET is the largest execution time of a program for any input and any initial execution state of the hardware platform. In general, it is undecidable whether or not the WCET is finite, because it is undecidable whether or not a program terminates. Hence, the WCET can only be computed for certain simply structured programs. For realistic and general programs, it is usually practically impossible to compute the WCET. Instead, reliable upper bounds have

to be determined by sound methods. Such upper bounds are usually called estimated WCET (WCET_{EST}) values and should have at least two properties:

1. The bounds should be safe ($\text{WCET}_{EST} \geq \text{WCET}$).
2. The bounds should be tight ($\text{WCET}_{EST} - \text{WCET} \rightsquigarrow 0$).

If safe WCET guarantees for hard real-time systems are needed, static program analyses are used. At binary code level, such static analyzers estimate register values in order to identify loop counters, determine loop iteration counts, and extract hardware-specific states of a processor's caches and pipelines. The path analysis stage finally estimates a program's global WCET by finding that path within a program's Control-Flow Graph (CFG) that has the maximal WCET – the so-called Worst-Case Execution Path (WCEP). The length of this longest path is the sum of the products $T * C$ over all blocks along the path, where T denotes a block's maximum execution time and C represents the block's maximal execution count.

26.2.2 Energy Consumption and Thermal Behavior

These days, we are almost exclusively using electrical devices to process information. Unfortunately, the operation of known devices requires the conversion of electrical energy into thermal energy. There are various reasons for trying to keep the amount of dissipated electrical energy as small as possible. For example, we would like to keep the impact on global warming as small as possible and we would like to avoid high operating temperatures and too high current densities. Energy may be available only in limited quantities. For mobile systems, electrical energy has to be either carried around with the system (e.g., in the form of batteries) or harvested (e.g., by using solar cells).

Using the consumed energy as an objective or constraint is not easy, since the amount of consumed energy depends on many factors. There are essentially two ways of estimating this objective: estimation can be either based on measurements for real hardware or based on computer models. Measurements can provide very precise results but can be performed only for existing hardware. Models can be used also for non-existing hardware, but they are inherently less precise.

The thermal behavior is very much linked to the energy consumption: the conversion of electrical energy into thermal energy is a source of heating the system. Thermal modeling has to take the thermal resistance between the system and the environment as well as thermal capacities into account. Again, computer models as well as measurements can be used.

26.2.3 Quality of Service and Precision

Overall, embedded systems have to provide some service, e.g., controlling a physical behavior (such as braking a car), showing some video, or generating some functional information. Such a service can be of high quality or of a reduced

quality. For example, a video can have various signal to noise ratios and different timing jitters. Control loops may be needing different amounts of time to stabilize. For functional information, there may be a deviation between known precise results and a computed approximation. In the following, the different levels of service will be called Quality of Service (QoS), and we will consider the precision of some functional result as a special case.

26.2.4 Safety, Security, and Dependability

Embedded systems may have a direct impact on their physical environment. Therefore, if embedded systems fail to perform the intended service, the physical environment may be at risk. System failures can be caused by internal malfunctions of the system as well as by attackers compromising the system. As a result, safety and security of embedded systems are extremely important.

Due to the impact on the physical environment, dependability of embedded systems is also important. By dependability, we capture the fact that an initially correctly designed and manufactured system may fail due to some internal fault, e.g., a bit flip in memory. Various physical effects can lead to such faults. Shrinking dimensions of microelectronic circuits are known to increase the rate of such faults [6]. Hence, they will have to be considered more carefully in the future.

26.2.5 Further Constraints and Objectives

There are many more constraints and objectives which are relevant. These include size, cost, and weight or the availability of hardware platforms. Embedded systems may have to resist certain types of radiation and may need to be environmentally friendly disposable. Not all of these can be described in detail in this chapter.

The principle of Pareto optimality can be used to take design decisions in the presence of multiple objectives. For further information on multi-objective design space exploitation, refer to our textbook [29] and to "▸ Chap. 6, "Optimization Strategies in Design Space Exploration"" of this book.

26.3 Optimization Potential in the Memory System

Much optimization potential is available in the memory system, because small memories are faster and consume less energy per access than larger memories. This observation was already made very early by Burks, Goldstein, and von Neumann in 1946 [7]:

> Ideally one would desire an indefinitely large memory capacity such that any particular ... word ... would be immediately available – i.e. in a time which is ... shorter than the operation time of a fast electronic multiplier. ... It does not seem possible physically to achieve such a capacity. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

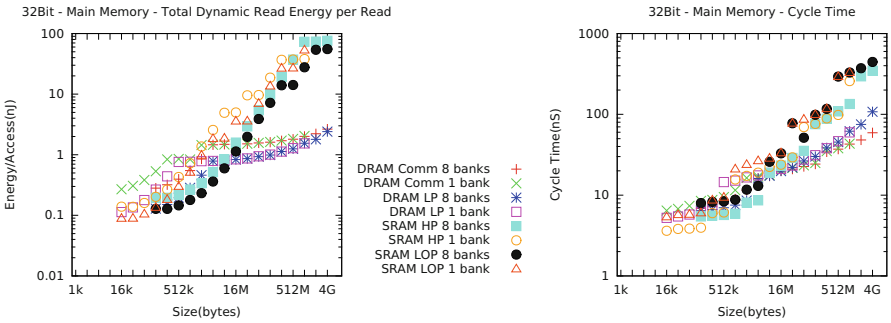


Fig. 26.1 Access times and energy consumption for DRAM and for SRAM

Figure 26.1 shows access times and energy consumptions per access for contemporary Dynamic Random-Access Memories (DRAMs) as well as for Static Random-Access Memories (SRAMs), for high-power and for low-power variants of these and for a single and eight banks. Numbers have been computed with CACTI [19]. Access times as well as energy consumptions vary by more than two orders of magnitude. Due to this, memory hierarchies have been introduced. Their key goal is to assign frequently accessed information to small and fast layers of the hierarchy such that, overall, the impression of a fast, energy-efficient, and still large memory is achieved on the average.

Small and fast memories thus act as buffers between main memory and the processor. For embedded systems, the architecture of these small memories has to be highly energy-efficient and must guarantee a predictable real-time performance. (More information on power and energy models can be found in ▸ Chap. 27, "Microarchitecture-Level SoC Design" of this book).

26.3.1 Caches

Let us briefly look again at some of the memories which were described in detail in ▸ Chap. 13, "Memory Architectures" of this book. *Cache*-based memory hierarchies are today's state of the art, because caches are fully transparent to the software running on a system. No code modification has to be done, since caches are hardware controlled. Caches are effective in exploiting *temporal locality* and *spatial locality*. The former means that particular memory locations will be accessed multiple times within a short period of time. The latter refers to the reference of contiguous memory locations over time.

N-way set-associative caches are organized as a matrix with N columns (usually called *ways*). During a memory access with a given address, the least significant bits of this address (i.e., its index bits) unambiguously identify the row of the cache matrix (usually called *set*) that potentially buffers the requested memory cell's contents. Within the selected cache set, the requested item can now reside in any of the N ways. Thus, the most significant address bits (the tag bits) are compared with the

tag bits buffered in all N ways of the selected set. If the tag bits match (cache hit), the requested memory cell is buffered in the cache and the data buffered in the identified way and set is returned to the processor. If no tag comparison matches (cache miss), the cache does not buffer the requested memory cell. A *replacement policy* is responsible for deciding which item to evict from the currently selected set if all ways of that set are currently occupied, but a new item shall be inserted in this set.

This architecture of set-associative caches combined with replacement policies enables a very high flexibility of caches so that they can autonomously adapt to varying memory access patterns issued by the processor. However, the drawbacks of caches are their large penalties in terms of the objectives introduced in Sect. 26.2. Caches exhibit a rather high-energy dissipation due to the additional memory required to store the tag bits and due to the hardware comparators performing the tag bit comparison for the currently selected cache set. Regarding real-time deadlines, caches are notorious for their inherent unpredictability. Depending on its replacement policy, it is hard, if not impossible, to predict during a static WCET analysis if a memory access results in a definite cache hit or miss. If a static WCET analyzer is uncertain about the cache's behavior, it has to assume the worst-case behavior of the cache which frequently leads to highly overestimated WCET_{EST} values.

Modern architectures support *cache locking*, i.e., cache cells are protected from being evicted by effectively partially disabling the replacement policy. This way, it is possible to predict access times of data or instructions that have been locked in the cache and to make precise statements about the cache's worst-case timing.

26.3.2 Scratchpad Memories

As an alternative to caches, small and "conventional" memories can be mapped into the processor's address space. These memories are frequently called Scratchpad Memories (SPMs) and differ from caches in that they are not operating autonomously in hardware. Instead, a simple address decoder decides whether a memory cell that is accessed by the processor is part of the SPM's address space or not, and the requested item is then fetched from the SPM or from some other memory.

Since SPMs completely lack tag memories and comparators, their energy efficiency is significantly higher than that of caches [5]. Furthermore, an access to the SPM always takes a constant time which is usually one clock cycle. As a consequence, varying memory access latencies due to cache misses or hits cannot occur in SPM-based architectures, thus rendering WCET estimates extremely tight and accurate. The drawback of SPMs is their lacking flexibility. Since they are unable to decide autonomously in hardware which items to buffer in and to evict from the SPM memory, there must be some software instance that assigns energy- or timing-critical parts of a program's code or data to the SPM. Frequently, this instance is the compiler that applies *scratchpad allocation techniques* and that determines a memory layout of a program such that it exploits the available SPM resources best. See ▸ Chap. 13, "Memory Architectures" of this book for a detailed comparison of caches and SPMs.

26.3.3 A Bound for Improvements

By how much can we improve memory references with respect to some objective on the average? Suppose that we are given two layers of the memory hierarchy and that memory m_i is closer to the processor and memory m_{i+1} further away from the processor. Suppose that a_i is the access time of memory m_i and a_{i+1} is the access time of m_{i+1}. Furthermore, let us assume that a fraction P of memory references to m_{i+1} can be replaced by references to m_i, leaving a fraction of $(1 - P)$ (the miss rate) of the memory references using m_{i+1}. Then, the average access time is

$$\text{average new access time} = P \cdot a_i + (1 - P) \cdot a_{i+1} \tag{26.1}$$

Let S be the ratio of access times (for available memory technologies, S can easily be in the order of 100):

$$S = \frac{a_{i+1}}{a_i} \tag{26.2}$$

The relative saving is

$$\begin{aligned}
\text{relative saving} &= \frac{\text{average old access time} - \text{average new access time}}{\text{average old access time}} \\
&= \frac{a_{i+1} - P \cdot a_i - a_{i+1} + P \cdot a_{i+1}}{a_{i+1}} \\
&= \frac{a_i \cdot S - P \cdot a_i - a_i \cdot S + P \cdot a_i \cdot S}{a_i \cdot S} \\
&= \frac{S - P - S + P \cdot S}{S} \\
&= P - \frac{P}{S}
\end{aligned} \tag{26.3}$$

The speedup of memory accesses can then be computed as follows:

$$\begin{aligned}
\text{speedup} &= \frac{\text{average old access time}}{\text{average new access time}} \\
&= \frac{a_{i+1}}{P \cdot a_i + (1 - P) \cdot a_{i+1}} \\
&= \frac{a_{i+1}}{P \cdot \frac{a_{i+1}}{S} + (1 - P) \cdot a_{i+1}} \\
&= \frac{1}{\frac{P}{S} + (1 - P)}
\end{aligned} \tag{26.4}$$

If S gets large, the first term in the denominator approaches zero and the improvement gets limited by the second term, $(1 - P)$. Hence, making S very large has a limited benefit when the miss rate cannot be made smaller. This effect is well known for caches. In practice, this means that the miss rate must be made as small as possible. Due to the problem of making the miss rate small, we have to make sure that almost every memory object can potentially be mapped to faster levels in the memory hierarchy. Excluding the stack, heap, or other memory objects from a mapping to fast memories would have a negative impact of the feasible speedup. In general, the goal of an ideal memory hierarchy is not always reached. Drepper has shown for the case of single-core systems that run times of programs can change by orders of magnitude if their working set exceeds the sizes of caches [13]. Hence, a clever use of memory hierarchies is already needed for single cores.

Equation (26.4) also corresponds to Amdahl's law [3], describing bounds for speedup by parallelization, where P is the fraction of the code which is parallelized and S is the speedup during parallel execution.

Equations (26.1) to (26.4) can also be generalized to capture effects for objectives other than access times. In particular, they can be applied to the case of modeling access energies and the resulting improvements.

26.3.4 Importance of Memory-Aware Load Balancing

For multi-core systems, an excessive amount of threads can lead to a lack of memory, as demonstrated, e.g., by Kotthaus and Korb [23]. Figure 26.2 (resulting

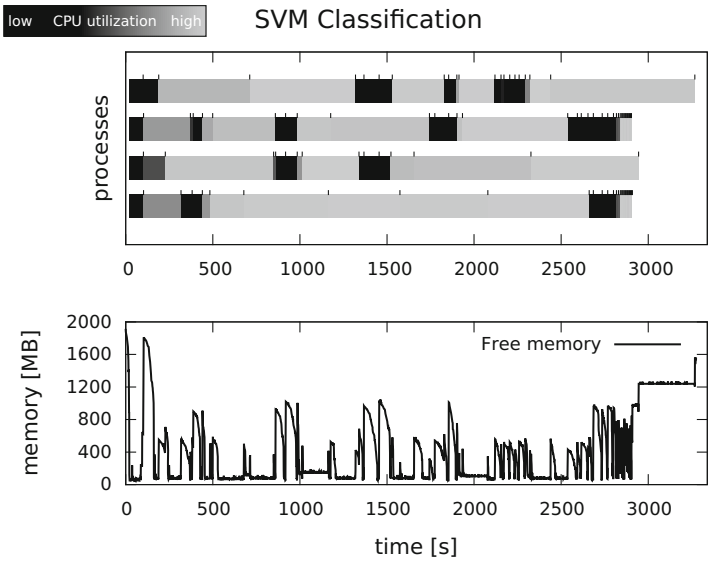


Fig. 26.2 SVM application on a four-core system: lack of free memory resulting in idling cores

from the experiments of Kottaus and Korb) shows profiling results for a Support Vector Machine (SVM) application running on a four-core system.

The application is programmed in the R language and is executed by version 3 of the R system running on a Lenovo L512 comprising an i5 processor. Due to memory-unaware allocation of cores, there are phases in which the system runs out of free memory. This happens even though load balancing of R is turned on. However, R is unaware of actual resource requirements. As a result, there is sometimes no free memory remaining. This is indicated by horizontal lines in the lower part of the diagram. Due to a too large number of processes, the system runs out of main memory and suffers from an increased swapping activity. This example demonstrates that memory-unaware allocation of computing resources results in wasting resources and cannot be efficient. We should care about required memory resources even for scenarios in which we address the programming of multi-core systems at a high level. Therefore, we will be looking at memory allocation in more detail in the remaining sections of this chapter.

26.4 Scratchpad Allocation Algorithms

26.4.1 Classification

In an earlier paper [5], we provided a detailed side-by-side comparison of caches and SPMs with respect to access times, energy consumptions, and silicon areas. A detailed comparison is also included in ▸ Chap. 13, "Memory Architectures" of this book.

In contrast to caches, SPMs must be explicitly managed by software. In the following, we classify the approaches to SPM management according to three dimensions:

- The type of allocation algorithm
- The type of architecture
- The optimization objective

We start by looking at allocation algorithms. SPM allocation algorithms can be classified into non-overlaying (or "static") and overlaying (or "dynamic") algorithms. For the first type of algorithms, memory objects are resident in the SPM during the entire lifetime of an application, whereas for the latter, objects are moved between the memories during run time.

26.4.2 Non-overlaying Allocation Algorithms

For the non-overlaying case, the optimization problem for energy or run-time optimization can be modeled as a Knapsack problem or as an Integer Linear Program (ILP).

Let i denote a memory object and let s_i denote its size. Let Δ_i denote the **saving** with respect to the considered objective if i is mapped to the SPM. The saving is the difference between the objective values for a mapping to some main memory and the SPM. Let S_{SPM} denote the size of the SPM. Let x_i be 1 if i is mapped to the SPM and 0 otherwise. Then, the following ILP model can be used to find an optimal mapping of memory objects to the SPM:

$$\text{Maximize} \quad \sum_i x_i * \Delta_i \tag{26.5}$$

Subject to

$$\sum_i x_i * s_i \leq S_{SPM} \tag{26.6}$$

Algorithms by Steinke [38] and by Verma [44] are examples based on such models. They are particular examples of hardware-aware compilation discussed in ▸ Chap. 25, "Hardware-Aware Compilation" of this book. In order to minimize the fraction $(1 - P)$ of "unimproved" memory references, most of our optimizations consider code and data references. For data references, global data can be easily taken into account. We have also considered stack variables. As a result, large savings (as computed by Equation (26.3)) have been observed.

We will demonstrate these for partitioned memories. Here, we have at our disposal J memories, each of them having an energy consumption e_j per access and a size S_j. Let n_i be the number of accesses to memory object i. A decision variable $x_{i,j}$ will be 1 if memory object i is mapped to memory j and 0 otherwise. Then, the following ILP model allows us to minimize the energy consumption:

$$\text{Minimize} \quad \sum_j e_j * \left(\sum_i x_{i,j} * n_i \right) \tag{26.7}$$

Subject to

$$\forall j \in J : \sum_i x_{i,j} * s_i \leq S_j \tag{26.8}$$

$$\forall i : \sum_j x_{i,j} = 1 \tag{26.9}$$

Figure 26.3 shows results for partitioned SPMs with 1 to 8 partitions.

For a single SPM, the savings (as computed by Equation (26.3) but indicated as a percentage, rather than a fraction) decrease when the SPM is larger than the working set of the application. For partitioned SPMs, the saving remains at the maximum, even for oversized SPMs. These savings refer to dynamic power consumption. Partitioned SPMs provide even larger advantages when leakage power is also taken into account.

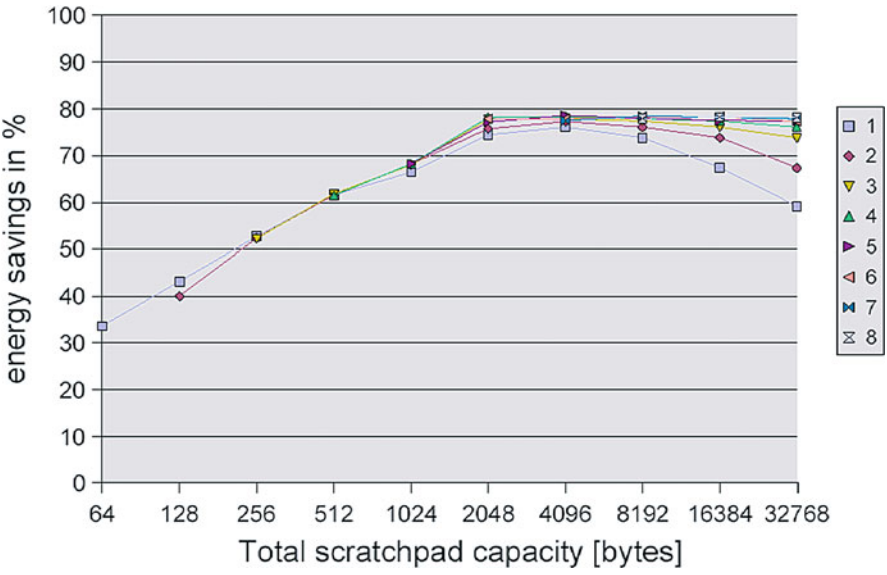


Fig. 26.3 Energy savings achieved by SPM allocation of a GSM application

26.4.3 Overlaying Allocation Algorithms

For overlaying algorithms, memory objects are migrated between different levels of the hierarchy. This migration can be either explicitly programmed in the application or inserted automatically. Overlaying algorithms are beneficial for applications with multiple hotspots, for which the code can be evicting each other. For overlaying algorithms, we are typically assuming that all applications are known at design time such that memory allocation can be considered at this time. Algorithms by Verma [44] and Udayakumararan et al. [41] are early examples of such algorithms.

Verma's algorithm starts with the CFG of the application to be optimized. For edges of the graph, Verma considers potentially freeing the SPM for locally used memory objects.

In Fig. 26.4, we are considering control blocks B1 to B10 and control flow branching at B2. We assume that array A is defined, modified, and used along the left path. T3 is only used in the right part of the branch. We consider potentially freeing the SPM so that T3 can be locally allocated to the SPM. This requires spill and load operations in potentially inserted blocks B9 and B10 (thin and dotted lines: potential inserts). Cost and benefit of these spill operations are then incorporated into a global ILP. Solving the ILP yields an optimal set of memory copy operations. For a set of benchmarks, the average reductions in energy consumption and execution time, compared to the non-overlaying case, are 34% and 18%, respectively. Blocks of code are handled as if they were arrays of data.

Udayakumararan's algorithm is similar, but it evaluates memory objects according to their number of memory accesses divided by their size. This metric is then

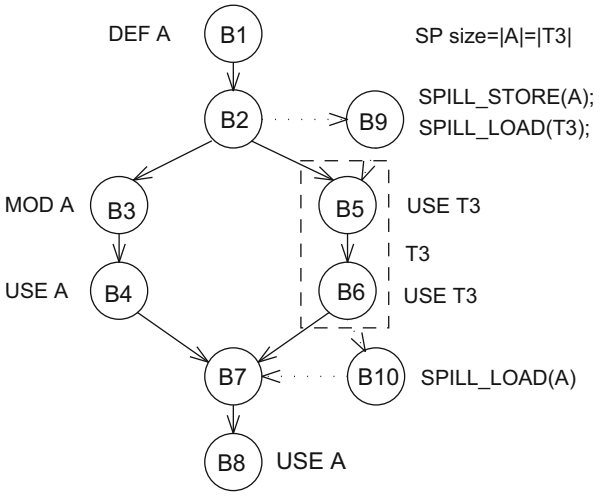


Fig. 26.4 Potential spill code

used to heuristically guide the optimization process. This approach can also take heap objects into account.

In more dynamic cases, the set of applications may vary during the use of the system. For such cases, dynamic memory managers are appropriate. Pyka [36] published an algorithm based on an SPM manager which is part of the operating system.

Egger et al. [9] proposed to exploit an existing Memory Management Unit (MMU) for dynamic replacements within the SPM. In his approach, code objects are classified into those that should potentially be moved into the SPM and those that should not. Potential SPM objects are then grouped into pages. Corresponding MMU entries are initially set to invalid. During execution, MMU exceptions are generated for accesses to SPM candidates not (yet) available in SPM. An exception handler is then invoked. The handler decides which memory objects to move into the SPM and which objects to move out. The approach is designed to handle code and is capable of supporting a dynamically changing set of applications. Unfortunately, the size of current SPMs corresponds to just a few entries in today's page tables, resulting in a coarse-grained SPM allocation.

Large arrays are difficult to allocate to SPMs. In fact, even a single array can be too large to fit into an SPM. The splitting strategy of Verma [16] is restricted to a single-array splitting. Loop tiling is a more general technique, which can be applied either manually or automatically [24]. Furthermore, array indexes can be analyzed in detail such that frequently accessed array components can be kept in the SPM [27].

Our explanations have so far mainly addressed code and global data. *Stack* and *heap data* require special attention. In both cases, two trivial solutions may be feasible: In some cases, we might prefer not to allocate code or heap data to the

SPM at all. Obviously, this would have an immediate effect on the bound for the achievable speedup as per Equation (26.4). In other cases, we could run stack [2] and heap size analysis [18] to check whether stack or heap fit completely into the SPM and, if they do, allocate them to the SPM.

For the heap, Dominguez et al. [12] proposed to analyze the liveness of heap objects. Whenever some heap object is potentially needed, code is generated to ensure that the object will be in the SPM. Objects will always be at the same address, so that the problem of dangling references to heap objects in the SPM is avoided. McIllroy et al. [30] propose a dynamic memory allocator taking characteristics of SPM into account. Bai et al. [4] suggest that the programmer should enclose accesses to global pointers by two functions $p2s$ and $s2p$. These functions provide conversions between global and local (SPM) addresses and also ensure a proper copying of memory contents.

For the stack, Udayakumararan et al. [41] proposed to use two stacks, one for calls to short functions with their stack being in main memory and one for calls to computationally expensive functions whose stack area is in the SPM. Kannan et al. [22] suggested to keep the top stack frames in the SPM in a circular fashion. During function calls, a check for a sufficient amount of space for the required stack frame is made. If the space is not available, old stack frames are copied to a reserved area in main memory. During returns from function calls, these frames can be copied back. Various optimizations aim at minimizing the necessary checks.

26.4.4 Supporting Different Architectures and Objectives

A second dimension in SPM allocation (in addition to the allocation type) is the architectural dimension. Implicitly, we have so far considered single-core systems with a single memory hierarchy layer and a single SPM. Other architectures exist as well. For example, there may be hybrid systems containing both caches and SPMs. We can try to reduce cache misses by selectively allocating SPM space in case of cache conflicts [8, 21, 48]. Also, we can have different memory technologies, like flash memory or other types of non-volatile RAM [45]. For flash memory, load balancing is important. Also, there might be multiple levels of memories. So far, we have just considered single-core processors. For multi-core systems, new tasks and options exist. SPMs can possibly be shared across cores. Also, there may be multiple memory hierarchy levels, some of which can be shared. Liu et al. [25] present an ILP-based approach for this.

A third dimension in SPM allocation is the objective function. So far, we have focused on energy or run-time minimization. Other objectives can be considered as well. Implicitly, we have modeled the average case energy consumption. The Worst-Case Energy Consumption (WCEC) is an objective considered, for example, by Liu [25]. Reliability and endurance are relevant for the design of reliable applications, in particular in the presence of aging [46]. It may also be necessary to avoid overheating of memories. From among other possible objectives, we will be looking at the WCET in the following sections.

26.5 WCET-Oriented Compiler Strategies

In contrast to simple optimization objectives like, e.g., energy consumption that can be modeled using single values like, e.g., Δ_i or e_j (cf. Sect. 26.4.2), the systematic reduction of WCET estimates is much more subtle due to the nature of the WCET. As already motivated in Sect. 26.2.1, the WCET of a program corresponds to the length of the longest path WCEP through a program's CFG. Thus, WCET-oriented optimizations must exclusively focus on those parts of the program that lie on the WCEP. The optimization of parts of the program aside the WCEP is ineffective, since this does not shorten the WCEP. Therefore, optimization strategies for WCET reduction must have detailed knowledge about the WCEP.

Unfortunately, this WCEP can be highly unstable in the course of an optimization. Consider the CFG of a function `main` in Fig. 26.5a, consisting of five basic blocks each of them having the indicated WCET values given in clock cycles. Obviously, the longest path through this CFG is `main`, `a`, `b`, and `c`. This WCEP, highlighted with solid arrows in Fig. 26.5a, has a WCET of 205 cycles. Assuming that some optimization is able to reduce `b`'s WCET from 80 down to 40 cycles, the CFG shown in Fig. 26.5b results from this optimization. As can be seen, the WCEP after optimization of `b` is `main`, `d`, and `c`. This example shows that the WCEP is very unstable during optimization – it can switch from one path within the CFG to a completely different one in the course of optimizations.

Thus, WCET-oriented and memory-aware compiler optimizations are faced with the challenges to always accurately model the current WCEP and to always be aware of possible WCEP switches. The following sections outline examples of WCET-oriented optimizations that exploit scratchpads and caches and that carefully consider WCEP switches. First, we present WCET-oriented SPM allocations to make the structural differences between memory-aware optimizations of energy dissipation (cf. Sect. 26.4) and of WCET estimates (cf. the following Sect. 26.5.1) evident. Next, we discuss cache locking optimizations in Sect. 26.5.2, followed by a presentation of cache partitioning for multitask systems in Sect. 26.5.3. Other approaches for timing models are explained in ▸ Chap. 19, "Host-Compiled Simulation" of this book. The importance of WCET-oriented optimizations for actual applications is stressed in Sect. 4 in ▸ Chap. 37, "Control/Architecture Codesign for Cyber-Physical Systems" of this book.

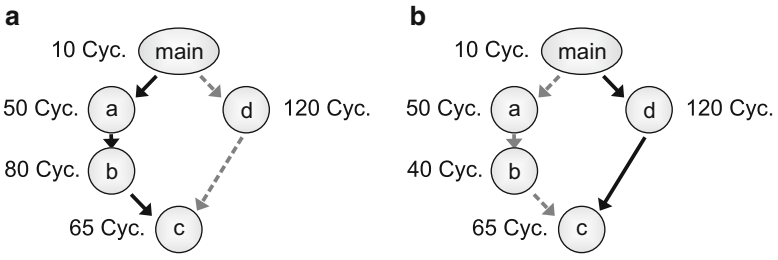


Fig. 26.5 (**a**) Original example CFG (**b**) Example CFG after optimization of `b`

26.5.1 WCET-Oriented Scratchpad Allocation

This section presents an ILP-based SPM allocation of program code that moves basic blocks statically onto the SPM [14, 39]. This is done under simultaneous consideration of possibly switching WCEPs by formulating ILP constraints that inherently model the longest path which starts at a certain basic block. The following equations use lowercase letters for ILP variables and uppercase letters for constants.

In analogy to the techniques presented previously in Sect. 26.4, the ILP also uses one binary decision variable v_i per basic block b_i of a program:

$$v_i = \begin{cases} 0 \text{ if basic block } b_i \text{ is assigned to } mem_{MAIN} \\ 1 \text{ if basic block } b_i \text{ is assigned to } mem_{SPM} \end{cases} \tag{26.10}$$

A scratchpad assignment is legal if the size of all basic blocks allocated to the SPM does not exceed the scratchpad's capacity. This property is ensured by adding in Equation (26.6) to the ILP again.

A block b_i of a function f causes some costs c_i, i.e., b_i's WCET_{EST} depending on whether b_i is allocated to main memory or to the SPM:

$$c_i = C^i_{MAIN} * (1 - v_i) + C^i_{SPM} * v_i \tag{26.11}$$

Constants C^i_{MAIN} and C^i_{SPM} model the WCET_{EST} values of b_i if it is executed from main memory or from the SPM, respectively. For reducible CFGs, an innermost loop l has exactly one back edge that turns it into a cyclic graph. Not considering this back edge turns l's CFG into an acyclic graph. This acyclic graph without l's back edge is denoted as $G_l = (V, E)$ here. Each node of G_l models a single basic block. Without loss of generality, there is exactly one unique exit node b^l_{exit} of loop l in G_l and one unique entry node b^l_{entry}. The WCET_{EST} w^l_{exit} of b^l_{exit} is set to the costs of b^l_{exit}:

$$w^l_{exit} = c^l_{exit} \tag{26.12}$$

The WCET_{EST} of a path from a node b_i (different from b^l_{exit}) to b^l_{exit} must be greater or equal than the WCET_{EST} of any successor of b_i in G_l, plus b_i's costs:

$$\forall b_i \in V \setminus \{b^l_{exit}\} : \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + c_i \tag{26.13}$$

Variable w^l_{entry} thus represents the WCET of all paths of loop l starting in b^l_{entry} if l is executed exactly once. To model several executions of l, all CFG nodes $v \in V$ of G_l are merged to a new super-node v_l. The costs of v_l are equal to l's WCET if executed once, multiplied by l's maximal loop iteration count C^l_{max}:

$$c_l = w^l_{entry} * C^l_{max} \tag{26.14}$$

Replacing a loop l by its super-node v_l may turn another loop l' of function f that immediately surrounds l into an innermost loop with acyclic CFG G'_l. Hence, Equations (26.12), (26.13), and (26.14) can be formulated analogously for l'. This way, the innermost loops of f are successively collapsed in the CFG so that ILP constraints that model f's control flow are created from the innermost to the outermost loops.

A program's WCEP can switch only at a block b_i with more than one successor because only there, forks in the control flow are possible. Since Equation (26.13) is created for each successor of b_i, variable w_i always reflects the WCET of any path starting from b_i – irrespective of which of the successors actually lies on the current WCEP. This way, Equation (26.13) realizes the implicit consideration of (switching) WCEPs in the ILP.

In analogy to the ILP modeling of loops, the WCET_{EST} of a program's function f is represented by the variable w^f_{entry} if basic block b^f_{entry} is F's unique entry point. Whenever a basic block b_i calls some function f, variable w^f_{entry} is added to w_i in Equation (26.13) in order to model the interprocedural control flow correctly.

Finally, an entire C program's WCET_{EST} is modeled by the ILP variable $w^{\texttt{main}}_{entry}$ that denotes the WCET_{EST} of the program's unique entry point `main`. To minimize a program's WCET by the ILP, the following simple objective function is thus used:

$$\text{Minimize } w^{\texttt{main}}_{entry} \tag{26.15}$$

Furthermore, our ILP includes many additional constraints that take care of adjusted branch instructions making sure that a basic block located in main memory can still branch to a successor placed onto the SPM, and vice versa [14]. The discussion of these branching related constraints is omitted here for the sake of brevity.

This structure of the ILP can also be used to allocate global variables of a program onto the SPM. The main difference between the SPM allocation of code as described by Equations (26.10)–(26.15) and that of data objects is the cost modeling part. A binary variable x_j per data object d_j of a program specifies whether to allocate it to the SPM or not:

$$x_j = \begin{cases} 0 \text{ if data object } d_j \text{ is assigned to } mem_{MAIN} \\ 1 \text{ if data object } d_j \text{ is assigned to } mem_{SPM} \end{cases} \tag{26.16}$$

Here, the scratchpad capacity constraint (26.6) is simply formulated over the decision variables and sizes of the allocatable data objects. Again, each basic block b_i of a program causes some costs c_i. For the SPM allocation of data, these costs c_i reflect b_i's WCET_{EST} depending on whether the data objects accessed by b_i are put in main memory or in the SPM:

$$c_i = C_i - \sum_{d_j \in \text{ data objects}} G_{i,j} * x_j \tag{26.17}$$

Here, C_i denotes b_i's WCET_{EST} if all data objects accessed by b_i are placed in main memory. $G_{i,j}$ is a constant that denotes the WCET reduction that b_i exhibits

if data object d_j is put on the SPM. All other constraints (26.12), (26.13), (26.14), and (26.15) of the SPM allocation of program code that model the structure of a program's CFG remain unchanged when allocating data to the SPM.

Both ILP models are fully integrated into the WCET-aware C Compiler (WCC) [15, 47]. Due to the integration of a static WCET analyzer into the compiler, the constants C^i_{MAIN}, C^i_{SPM} and C_i used in Equations (26.11) and (26.17) are determined fully automatically. The sizes of basic blocks, data objects and SPM memories as well as the gain $G_{i,j}$ from Equation (26.17) are determined using WCC's processor-specific low-level intermediate representation. The maximal loop iteration counts C^l_{max} used in Equation (26.14) stem from WCC's polyhedral loop analyzer [26] or from user annotations. The WCC compiler infrastructure allows to generate the ILPs, their solution using IBM's `cplex` solver and the final memory allocation of the binary executable code in a fully automated fashion without any user intervention.

The following paragraphs show some experimental results that illustrate the WCET_{EST} reductions that can be achieved by these two SPM allocations of program code and of data. Experiments have been performed for an Infineon TriCore TC1796 processor that features a 47 kB code SPM and a separate 40 kB data SPM that are both accessible within one clock cycle. The processor's main memory has an access latency of six clock cycles. WCET analyses were performed using the static timing analyzer aiT [1]. All results are generated using WCC's optimization level *-O2* so that our SPM allocations were applied to already highly optimized code.

We applied our SPM allocation of program code to 73 different real-life benchmarks. Code sizes range from 52 bytes up to 18 kB. Since these code sizes are much smaller than the totally available SPM size, we artificially limit the available SPM space for benchmarking. For each benchmark, SPM sizes of 10%, 20%, ..., 100% of the benchmark's code size were used. Figure 26.6 shows the WCET estimates of all benchmarks produced by the WCET analyzer aiT that result from our SPM allocation as a percentage of the WCET_{EST} when not using the program SPM at all. The bars in the diagram represent the average values over all 73 benchmarks. As can be seen, steadily decreasing WCET_{EST} values were observed for increasing SPM sizes. Already for tiny SPMs with a capacity of 10% of a benchmark's code size, WCET_{EST} decreases to 92.6% compared to the case

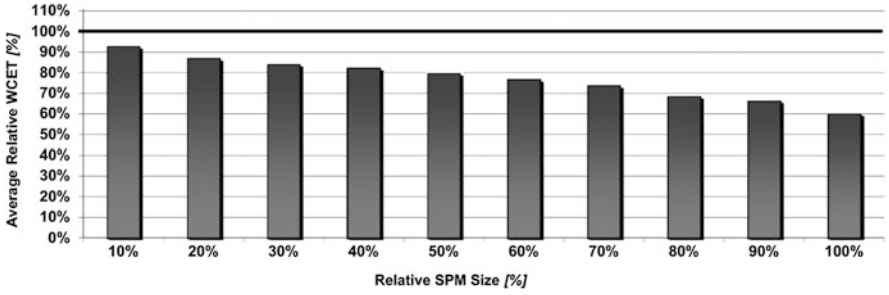


Fig. 26.6 Average relative WCET_{EST} values after WCET-oriented SPM allocation of code

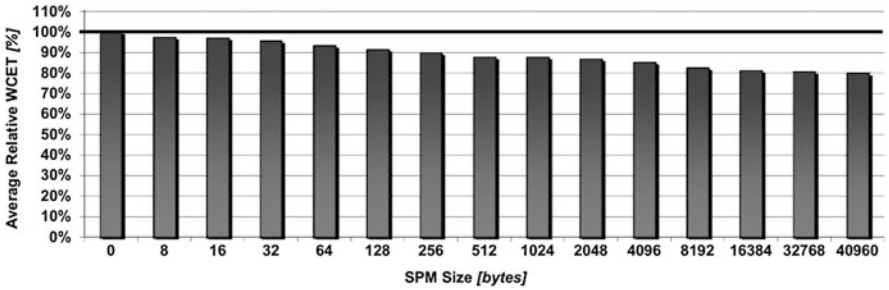


Fig. 26.7 Average relative WCET_{EST} values after WCET-oriented SPM allocation of data

when not using the SPM at all. For SPMs large enough to hold entire benchmarks, an average WCET_{EST} of only 60% of the original WCET was obtained. Thus, the achieved savings range between 7.4% and 40%. Our SPM allocation of program code potentially changes the benchmarks' code sizes due to the insertion of adjusted jump instructions in order to keep the control flow correct. It turned out that these changes are negligible – we observed a maximal code size increase by 128 bytes for our benchmarks. On average over all 73 benchmarks, code sizes increased by 0.02%.

Figure 26.7 shows the results of our SPM allocation of data averaged over all benchmarks that contain global data. The x-axis represents varying SPM sizes in absolute values. Again, we observed that WCET_{EST} decreased steadily for increasing data SPM sizes. Already for SPMs of only 8 bytes size, average WCET estimates over all benchmarks were reduced by 2.6%. For the real TriCore architecture with its 40 kB data SPM, average overall savings of 20.2% were achieved. The run-time complexity of both ILP-based SPM allocations is negligible in practice. ILP solving times of at most two CPU seconds were observed on an Intel Xeon at 2.4 GHz.

Both SPM allocations for code and data assume constant values to represent WCET values of basic blocks depending on the actual memory allocation (cf. Equations (26.11) and (26.17)). This in turn implies that the access latencies of the memories are also assumed to be constant like the six clock cycles for main memory accesses considered in this section. However, if Flash memory is used as main memory, its access latencies can vary, since Flash memory is organized in blocks and consecutive accesses within one block are faster than the six clock cycles used here. This behavior of Flash memories has no effect on the SPM allocation of code, since Equation (26.11) uses WCET values provided by a static timing analyzer that is inherently aware of the varying access latencies of Flash memories. The SPM allocation of data uses a constant gain $G_{i,j}$ for data memory accesses in Equation (26.17) which relies on the assumption of constant access latencies. Thus, this SPM allocation could take suboptimal allocation decisions so that its objective function from Equation (26.15) does not optimally minimize the global WCET of a program. However, since all WCET estimates used to generate Fig. 26.7 were solely obtained by aiT with its built-in support for Flash memories, our results can be considered safe, and the savings depicted in Fig. 26.7 are considerable despite of

the conservative ILP model. We expect that the additional WCET reductions that could potentially be achieved by an improved ILP model considering block Flash accesses are marginal and not worth the effort.

26.5.2 Static Instruction Cache Locking

It is worthwhile mentioning that the structure of the ILP presented in the previous section is very general and flexible so that it can be employed to realize other memory-oriented optimizations beyond SPM allocation. The key difference between SPM allocation and cache locking is the granularity of the items to be allocated to the SPM or cache, respectively. In the case of SPMs, basic blocks or global variables of arbitrary size are candidates for memory allocation – cf. Equations (26.10) and (26.16). In contrast, the granularity of items that can be locked into a cache is defined by the cache's hardware architecture and its lockdown scheme.

For example, the ARM926EJ-S architecture supports way-based instruction cache locking which means that only complete columns of an N-way set-associative cache (cf. Sect. 26.3) can be locked. For an N-way set-associative cache with a total capacity of S_{CACHE} bytes, each way comprises S_{WAY} bytes:

$$S_{WAY} = S_{CACHE}/N \tag{26.18}$$

For a size B of a cache block given in bytes, the number of lines L per cache way is

$$L = S_{WAY}/B \tag{26.19}$$

Loading content from the main memory and locking it into a single cache line causes some architecture-specific but constant costs C_{LINE}. Thus, the costs for locking a complete way consisting of L lines are

$$C_{WAY} = L * C_{LINE} \tag{26.20}$$

Due to the modulo addressing of caches, memory addresses with $addr \bmod S_{WAY} \equiv 0$ are mapped to the beginning of a cache way. Thus, the main memory can be divided into memory blocks mb with a size of S_{WAY} bytes each such that each block can be entirely locked into a single cache way. This partitioning into blocks of size S_{WAY} is then applied to a program to be optimized by our cache locking approach – these blocks $mb_1, \ldots, mb_m$ denote candidates for cache locking. Thus, the ILP for instruction cache locking includes binary decision variables per memory block mb_j:

$$y_j = \begin{cases} 0 \text{ if memory block } mb_j \text{ remains unlocked} \\ 1 \text{ if memory block } mb_j \text{ is locked into the instruction cache} \end{cases} \tag{26.21}$$

An N-way set-associative cache can keep copies of up to N such memory blocks at the same time, since ways can only be locked in their entirety. Thus, an ILP constraint needs to ensure that the size of the contents locked into the cache does not exceed the cache size:

$$\sum_{j=1}^{m} y_j \leq N \tag{26.22}$$

As already shown in Sect. 26.5.1, each basic block b_i of a program causes some costs c_i. The WCET estimate of b_i if it is executed from main memory is denoted by the constant C^i_{MAIN} while C^i_{CACHE} represents its Worst-Case Execution Time if the block is locked into the cache. Given the size S_i of each basic block b_i and its start address in main memory, it is easy to determine the number of bytes $S_{i,j}$ of b_i that are part of memory block mb_j. Then, the potential WCET_{EST} reduction of b_i in clock cycles $R_{i,j}$ if parts of it are executed from the cache due to a lockdown of mb_j is:

$$R_{i,j} = \frac{S_{i,j}}{S_i} * (C^i_{MAIN} - C^i_{CACHE}) \tag{26.23}$$

In the ILP for instruction cache locking, the costs c_i reflect b_i's WCET_{EST} depending on whether memory objects that b_i is part of are locked into the cache:

$$c_i = C^i_{MAIN} - \sum_{j=1}^{m} y_j * R_{i,j} \tag{26.24}$$

Using these basic block costs c_i, the constraints from Equations (26.12), (26.13), (26.14) that model the structure of a program's CFG can, again, be reused without any further modification in order to realize the ILP for cache locking.

In analogy to Sect. 26.5.1, the WCET_{EST} of a complete C program is represented by the ILP variable $w^{\texttt{main}}_{entry}$. However, static instruction cache locking as presented here involves some overhead in terms of WCET_{EST}, since some newly inserted code for loading and locking contents into the cache needs to be executed in the very beginning of function `main`. Thus, the objective function of the ILP for instruction cache locking that has to be minimized now models the WCET_{EST} of the complete program including this lockdown overhead [34]:

$$\text{Minimize } w^{\texttt{main}}_{entry} + \sum_{j=1}^{m} y_j * C_{WAY} \tag{26.25}$$

This ILP model is again fully integrated and automated within the WCC compiler [15,47]. An evaluation has been carried out for an ARM926EJ-S processor that features a 16-kB large instruction cache with 32-byte line size, Least-Recently Used (LRU) replacement, and a configurable associativity of 2 or 4. Content can be accessed from the cache within one clock cycle, while main memory accesses

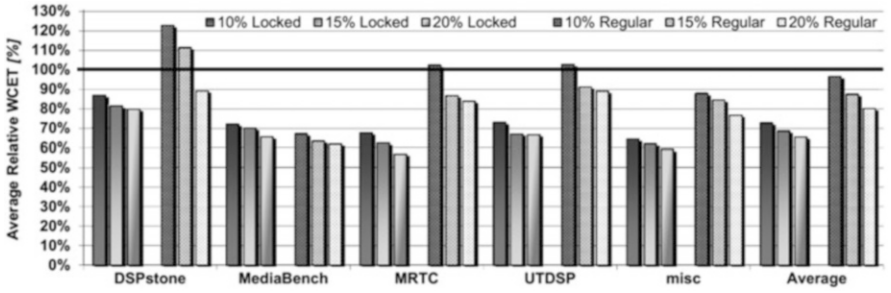


Fig. 26.8 Average relative $WCET_{EST}$ values after WCET-oriented locking of 2-way set-associative instruction caches

take six cycles. The instruction cache supports way-based locking as described in this section. Loading and locking a single cache line of 32 bytes takes $C_{LINE} = 47$ clock cycles. The assumption of constant values for main memory access latencies and thus for the cache line locking overhead C_{LINE} has already been discussed at the end of Sect. 26.5.1. In the context of the instruction cache locking presented here, the imprecision of the ILP model is again considered marginal, since the constant locking overhead contributes exactly once to a benchmark's overall WCET, because the locking code is executed exactly once during the system startup phase.

Our cache locking optimization has been evaluated using 100 different real-life benchmarks from various commonly used benchmarking suites (DSPstone, MediaBench, MRTC, UTDSP, and some benchmarks from miscellaneous sources). For our evaluations, we artificially limited the cache sizes to 10%, 15%, and 20% of a benchmark's overall code size.

Figure 26.8 depicts the results of our static instruction cache locking scheme if applied to an architecture with a 2-way set-associative cache. The bars of the diagram show the WCET estimates that result from our cache locking as a percentage of the $WCET_{EST}$ when executing the benchmarks without any cache. The figure shows average results over all used benchmark suites for the sake of readability. Per benchmark suite, detailed results are given for ILP-based cache locking as well as for a freely operating instruction cache without any locking. All locking-based results include the overhead for loading and locking blocks into the cache prior to a benchmark's execution. As can be seen from Fig. 26.8, ILP-based cache locking leads to maximal overall $WCET_{EST}$ reductions of 35.4% (misc benchmarks) for very small caches with a capacity of 10% of a benchmark's code size. For caches of size 15% and 20%, respectively, the maximally achieved $WCET_{EST}$ reductions increase up to 37.7% (misc benchmarks) and 43.1% (MRTC). The maximal improvements achieved by a regularly operating cache without locking amount to 32.6%, 36.3%, and 37.8% for the MediaBench suite and caches of sizes 10%, 15%, and 20%, respectively. Interestingly, our proposed cache locking always outperforms the regular caches except for MediaBench. Due to the static nature of the cache locking approach described here, the originally dynamic behavior of the cache gets lost. MediaBench exhibits a number of computation kernels that cannot be locked simultaneously into the size-restricted cache. In contrast, the regularly

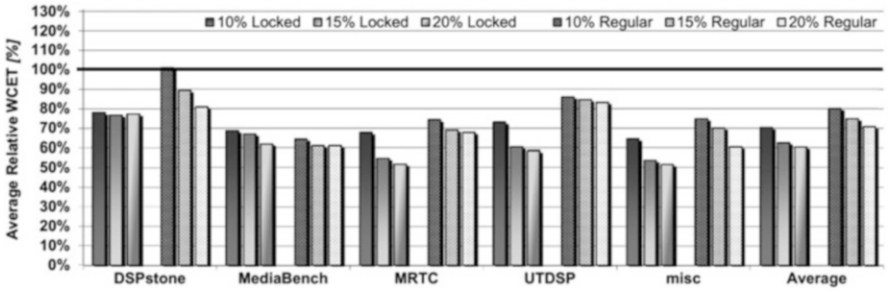


Fig. 26.9 Average relative $WCET_{EST}$ values after WCET-oriented locking of 4-way set-associative instruction caches

operating cache can exchange the content during run time and adapt better to the characteristics of these benchmarks. Partial cache locking as described in [11] would be an alternative for such cases. For all other benchmark suites, the $WCET_{EST}$ values resulting from our cache locking are significanly lower than those obtained by a regular cache. On average over all 100 considered benchmarks, locking leads to improvements of 27.1%, 31.2%, and 34.3% for 10%, 15%, and 20% large caches, respectively, while the unlocked caches of the same sizes only show improvements of 3.3%, 12.4%, and 19.6%, respectively.

The same trends were observed for a 4-way set-associative cache (cf. Fig. 26.9). Compared to the previous case with associativity of 2, the regular and unlocked 4-way set-associative cache achieves much better results due to its higher degree of freedom in which way to store some blocks: here, maximal $WCET_{EST}$ reductions of 35.5%, 38.9% (MediaBench), and 39.5% (misc) were achieved. However, the locked cache still outperforms the unlocked one except for MediaBench. Cache locking leads to maximal improvements of 35.4%, 46.1%, and 48.3% (misc benchmarks) for caches of size 10%, 15%, and 20%, respectively. On average over all benchmarks, locking the 4-way set-associative cache improves $WCET_{EST}$s between 29.5% (10% cache size) and 39.6% (20% cache size) while the regular unlocked cache only achieves reductions from 19.8% (10% cache size) up to 29.2% (20% cache size).

Locking of content into data caches could be done in a similar fashion as described here. A first approach on static data cache locking using compile-time cache analysis was originally proposed in [42].

26.5.3 Instruction Cache Partitioning for Multitask Systems

While the techniques described so far are effective in reducing the $WCET_{EST}$ of a single program, today's systems are often multitask systems where different programs are preempted and activated by a scheduler. For such multitask systems, caches are an even larger source of timing unpredictability as compared to single-task systems (cf. Sect. 26.3), because interrupt-driven schedulers lead to unknown points of time where task preemptions and context switches may happen.

Furthermore, it may happen that one task evicts cache contents belonging to some other task so that this other task exhibits additional cache misses if it resumes its execution. Finally, it is also unknown at which address the execution of a preempted task continues; hence it is unknown which cache set is accessed and eventually evicted next. Recent work on Cache-Related Preemption Delay (CRPD) analysis tries to incorporate scheduling and task preemption into timing analysis. But since the behavior of a cache in preemptive multitask systems cannot be predicted with 100% accuracy, the resulting WCET estimates are often highly overestimated, or some scheduling policies are not analyzable at all.

Cache partitioning is a technique to make the I-cache behavior perfectly predictable even for preemptive multitask systems. Here, the cache is divided into partitions of different sizes, and each task of a multitask system is assigned to one of these partitions. This partitioning is done such that each task can only evict entries from the cache that belong to its very own partition. By construction, a task can never evict cache contents of other tasks. As a consequence, multiple tasks do not interfere with each other any longer w.r.t. the cache during context switches. This allows to apply static WCET analyses for each individual task of the system in isolation. The overall WCET_{EST} of a multitask system using partitioned caches is then composed of the WCET_{EST} values of the single tasks given a certain partition size, plus the overhead required for scheduling and context switching.

Cache partitioning can be realized fully in software and thus does not require any support by the underlying cache hardware, as opposed to cache locking as presented in the previous Sect. 26.5.2. For this purpose, the code of each task has to be scattered over the main memory's address space in a way that it only uses such memory addresses that map to those cache sets that belong to the task's partition. Thus, a task's executable code is split into many chunks which are stored in non-consecutive regions in main memory. To make sure that a task's control flow remains correct after splitting it into chunks, additional jump instructions between these chunks need to be inserted. The generation of these chunks in the executable code can be done easily by the linker if a dedicated linker script describing this scattering and the different chunks is provided.

The remaining challenge consists of determining a partition size per task such that a multitask system's overall WCET_{EST} is minimized. In the following, preemptive round-robin scheduling of tasks is assumed and the period P_i of each task $t_i \in \{t_1, \ldots, t_m\}$ is known a priori. The length of the entire system's hyper-period is equal to the least common multiple of all tasks' periods P_i. The schedule count H_i then reflects the number of times that each task t_i is executed within a single hyper-period. Furthermore, a couple of n possible cache partition sizes $S_j \in \{S_1, \ldots, S_n\}$ measured in bytes is given beforehand.

WCET-aware software-based cache partitioning is modeled inside the WCC compiler using integer linear programming again [35]. A binary decision variable $z_{i,j}$ is used to model whether task t_i is assigned to a partition of size S_j:

$$z_{i,j} = \begin{cases} 0 \text{ if task } t_i \text{ is not assigned to a partition of size } S_j \\ 1 \text{ if task } t_i \text{ is assigned to a partition of size } S_j \end{cases} \tag{26.26}$$

The following constraints ensure that each task is assigned to exactly one partition:

$$\forall \text{ tasks } t_i \in \{t_1, \ldots, t_m\} : \sum_{j=1}^{n} z_{i,j} = 1 \tag{26.27}$$

In analogy to the SPM allocations presented in Sect. 26.5.1, the cache capacity constraint is given by:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} z_{i,j} * S_j \leq S_{CACHE} \tag{26.28}$$

It is assumed here that the WCET_{EST} $C_{i,j}$ of each task t_i if executed once using a cache partition of each possible size S_j is given a priori. This is achieved by performing a WCET analysis of each task for each partition size before generating the ILP for software-based cache partitioning. A task t_i's WCET_{EST} c_i depending on the partition size assigned to the task by the ILP can thus be expressed as:

$$\forall \text{ tasks } t_i \in \{t_1, \ldots, t_m\} : c_i = \sum_{j=1}^{n} z_{i,j} * C_{i,j} \tag{26.29}$$

The objective function of the ILP models the WCET_{EST} of the entire task set for one hyper-period. This overall WCET estimate to be minimized is thus defined by:

$$\text{Minimize } \sum_{i=1}^{m} H_i * c_i \tag{26.30}$$

Figure 26.10 shows the WCET estimates achieved by cache partitioning for three different benchmark suites. Since no multitask benchmark suites currently exist, randomly selected task sets from single-task benchmark suites were used.

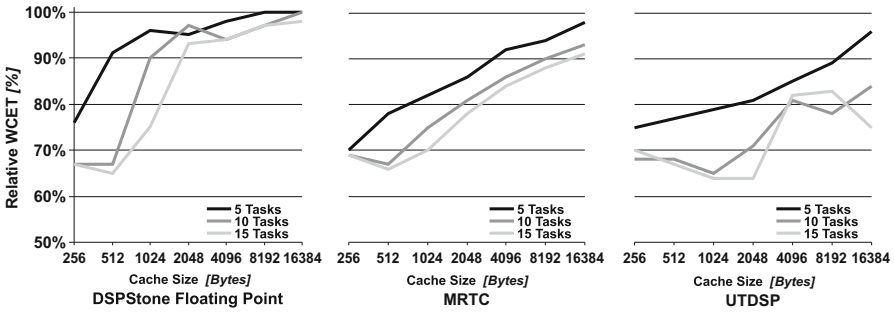


Fig. 26.10 Average relative WCET_{EST} values after cache partitioning for multitask systems

Figure 26.10 shows results for task sets consisting of 5, 10, and 15 tasks, respectively. Each individual point in the figure's curves denotes the average value over 100 randomly selected task sets of a certain size. Benchmarking was done for an Infineon TriCore TC1796 processor with instruction cache sizes ranging from 256 bytes up to 16 kB. An access to the cache requires one clock cycle, accessing the main memory takes six cycles. All results are given as a percentage, with 100% corresponding to the $WCET_{EST}$ values achieved by a standard heuristic that uses a partition size per task which depends on the task's code size relative to the code size of the entire task set.

As can be seen, substantial $WCET_{EST}$ reductions of up to 36% were obtained. In general, WCET savings are higher for small caches and lower for larger caches. For DSPstone, $WCET_{EST}$ reductions between 4% and 33% were achieved. For the MRTC benchmarks, an almost linear correlation between $WCET_{EST}$ reductions and cache sizes was observed, with maximal WCET savings of 34%. For the large UTDSP benchmarks, $WCET_{EST}$ reductions of up to 36% were finally observed. In most cases, larger task sets exhibit a higher optimization potential so that cache partitioning achieves higher $WCET_{EST}$ improvements as compared to smaller task sets.

Software-based instruction cache partitioning as described in this section can be used for any processor and does not require any hardware support. However, hardware cache partitioning could be advantageous if dynamic repartitioning and adaptation of partition sizes at run time are desired. Such a dynamic partitioning scheme is difficult to realize in software, because it involves relocating the scattered code in memory at run time. As mentioned previously, additional jump instructions need to be added to the tasks' code in order to keep its control flow correct. The additional overhead contributed by these jumps is obviously the larger, the smaller the considered cache sizes are. For tiny caches of only 256 bytes, the overhead due to the additional jumps lies between 10% and 34% of the benchmark's total $WCET_{EST}$ for UTDSP and MRTC, respectively. For 16 kB large caches, the overhead lies between 1% and 2%.

A combination of partitioning and locking for data caches has been proposed in [43]. The authors use dynamic cache locking, static cache analyses and cache partitioning to ensure that all intratask conflicts, and consequently, memory access times, are exactly predictable.

26.6 Trade-Off Between Energy Consumption, Precision, and Run Time

26.6.1 Memory-Aware Mapping with Optimized Energy Consumption and Run Time

Thiele et al. designed the DOL tool for the optimized mapping of applications to multi-processor systems on a chip (MPSoCs) [40]. The original system is unaware of the sizes of the involved memory systems. Jovanovic modified this

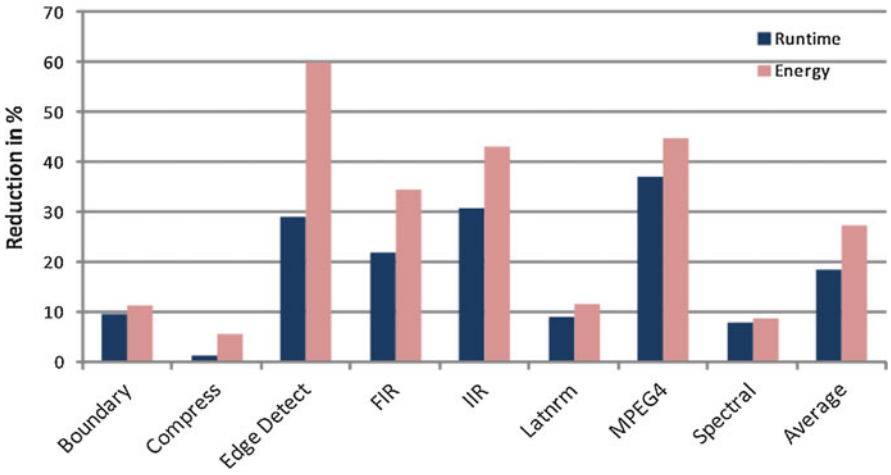


Fig. 26.11 Reduction in run time and energy achieved by run-time minimization

system such that the characteristics of available memories are also taken into account [20]. Also, the modified tool accepts general C-programs as input, instead of Kahn process networks. Input programs are generated by automatic parallelization [10]. Communication is based on First-In First-Out (FIFO) buffers. Two separate ILP models minimize either the energy consumption or the run time. The model minimizing the execution time includes access times of memories as well as expected execution times for the processors. Optimizations exploit available fast on-chip memories.

Figure 26.11 shows the reduction in run time and energy consumption for an ILP system minimizing run time. The baseline is an ILP-based mapping using run time as its objective.

In this case, the execution platform comprises four processors, each equipped with local data and instruction level-1 SPMs and a larger local level-2 memory. Furthermore, the platform includes a global shared memory which is used for communication. On average, memory awareness results in a reduction of the run time by 18% and of the energy by 27%.

Figure 26.12 shows the corresponding reduction for an ILP system minimizing the energy consumption. The baseline is an ILP-based mapping using energy as its objective. Compared to the results for run-time minimization, average run time is increased by 28%.

Both figures prove by means of an example that memory awareness allows a reduction of objectives run time and energy consumption. Also, minimization of run time does not automatically minimize energy consumption and vice versa, despite time being one variable in the computation of the energy consumption.

In a similar way, a trade-off between QoS and timeliness of results can be considered. For example, we can introduce qualifiers indicating whether or not

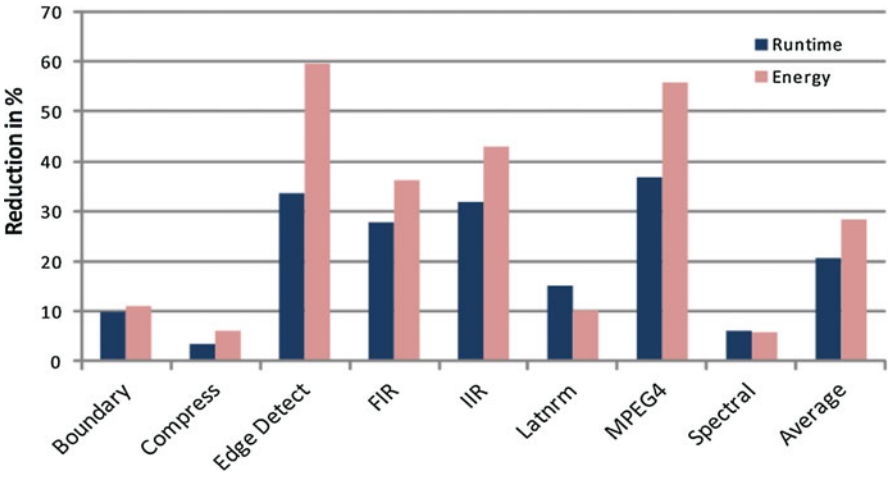


Fig. 26.12 Reduction in run time and energy achieved by energy minimization

variables should be allocated to reliable memory [37]. For variables not requiring reliable memory reads, we can skip error correction in the interest of timeliness of results.

26.6.2 Optimization for Three Objectives for the PAMONO Virus Sensor

In this section, we would like to demonstrate by means of an example how trade-offs between several objectives can be considered in such a way that reliable energy estimates are used. As an example, we will use a CPS for the detection of biological viruses based on Plasmon-Assisted Microscopy of Nano-Objects (PAMONO). The overall structure of the system can be seen in Fig. 26.13.

The sensor system includes an optical prism. On one of the sides (at the top in Fig. 26.13), there is a very thin gold layer covered with antibodies. Laser light entering through the second side of the prism, illuminating the backside of the gold layer and leaving through the third side is captured by a video camera. This prism is attached to a flow cell where the samples are applied. A pipe can be used to pump streams of gas or liquids across the gold layer. In case the stream contains viruses, they get stuck onto the gold layer with a certain probability. In case this happens, reflectivity of light reflected on the other side of the gold layer is affected and captured by the camera. Due to a resonance effect, the change is visible even when the size of the viruses is smaller than the wavelength of light. Real-time diagnosis of viruses like chicken-flu is among the potential applications.

However, due to the small dimensions of the camera sensor, video streams contain a significant amount of noise. A sophisticated image processing pipeline is needed in order to achieve a good detection quality. Figure 26.14 shows the pipeline

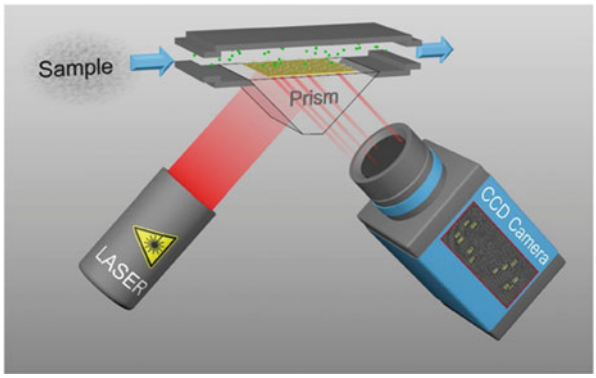


Fig. 26.13 Overall structure of the PAMONO virus sensor [33]

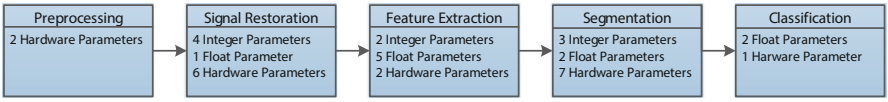


Fig. 26.14 Image processing pipeline to detect viruses. Listed parameters are modified by the GA to optimize the performance [33]

used in our research. In contrast to the previous section on WCET optimization, the application has soft real-time requirements.

In the pre-processing step, 16-bit gray-scale images are copied to the Graphics Processing Unit (GPU) and converted to floating-point arrays. In the next step, constant background noise is removed, and the signal of attaching virus is restored based on a sensor model of the PAMONO sensor. This step includes parameters which can be optimized for the best noise reduction under given circumstances. Various per-pixel and per-polygon features are computed during the feature extraction step. Per-pixel features describe the degree of membership to pixel classes representing virus adhesion. Per-polygon features perform the same function for polygons and their membership to polygon classes representing virus adhesion. During feature extraction, parameters comprise detection thresholds and parameters to switch between different feature extraction algorithms. Segmentation parameters control the way in which polygons are created and the way in which extracted features per pixel are combined to features per polygon. False classifications are minimized by appropriate classification parameters. The virus detection quality is measured with the F_1 score. This score is defined as the harmonic mean of the precision p and recall r:

$$F_1 = 2\frac{p \cdot r}{p + r} \tag{26.31}$$

with

$$\text{precision } p = \frac{TP}{TP + FP} \tag{26.32}$$

$$\text{and recall } r = \frac{TP}{TP + FN} \tag{26.33}$$

$$TP : \text{true positives}$$

$$FP : \text{false positives}$$

$$FN : \text{false negatives}$$

One of the goals of our research was to demonstrate that the overall system can be downsized from a PC-based environment such that it can be operated even in environments with limited compute performance and power availability. For demonstration purposes, we selected the Odroid-XU3 [17] platform as an execution platform.

The XU3 contains two powerful multi-core processors with four cores each and a GPU resulting in a performance matching the needs of our application. The overall structure is shown in Fig. 26.15. Also and very importantly, it provides facilities for measuring the currents for all the cores and the memory. In this way, we can get around the problem of the limited precision of computer-based energy models. This allows considering energy during the optimization of the mapping of our application to the cores and the GPU. Unfortunately, the Odroid XU3 is superseded by the Odroid XU4 platform, which does not have this facility.

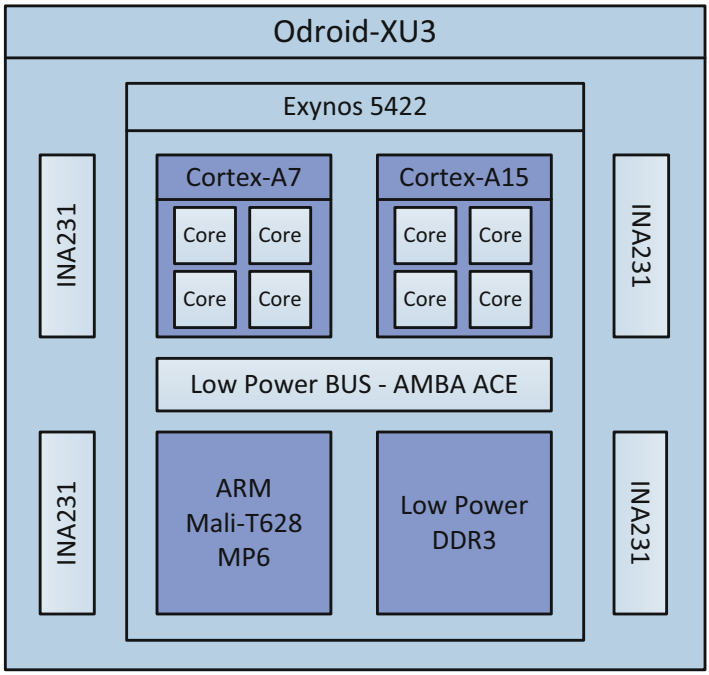


Fig. 26.15 Odroid execution platform [33]

A Genetic Algorithm (GA) is used to find Pareto-optimized design points considering execution times, energy consumption, and detection quality as objectives. (See ▸ Chap. 6, "Optimization Strategies in Design Space Exploration" of this book for a general discussion of Genetic Algorithm (GA)-based design space exploration. Other approaches for automatic parallelization and mapping to platforms can be found in ▸ Chaps. 28, "MAPS: A Software Development Environment for Embedded Multicore Applications" and ▸ 29, "HOPES: Programming Platform Approach for Embedded Systems Design" of this book.) The design space exploration is based on a heavily modified version of ECJ (Java-based Evolutionary Computation Research System) [28]. These modifications take parameter dependencies and parameter restrictions into account such that invalid parameter combinations are not generated. The evaluation of run times and energy consumption is based on the execution of the software on two available Odroid XU3 platforms concurrently as shown in Fig. 26.16. GPU; both Central Processing Units (CPUs) and memory energy consumptions are measured. Fitness results are averaged over several executions in order to remove jitter. Our energy measurement tool has been made publicly available [32].

An overview of the pipeline parameters is depicted in Fig. 26.14. Software parameters ranging from Boolean to restricted [0,1] floating-point values lead already to a large solution space. Beside pipeline parameters of the detection algorithm, several hardware parameters of our Odroid platform are considered by the optimization algorithm:

1. The used governor controlling operating parameters at run time (e.g., performance, powersave, interactive)
2. The frequency (200 MHz to 2 GHz in 100 MHz steps) of the Cortex-A15 core, if control is allowed by the governor
3. Work group sizes of all pipeline elements mapped to the Mali-T628 GPU
4. The memory allocation size for buffers storing among other things the detected polygons on the GPU

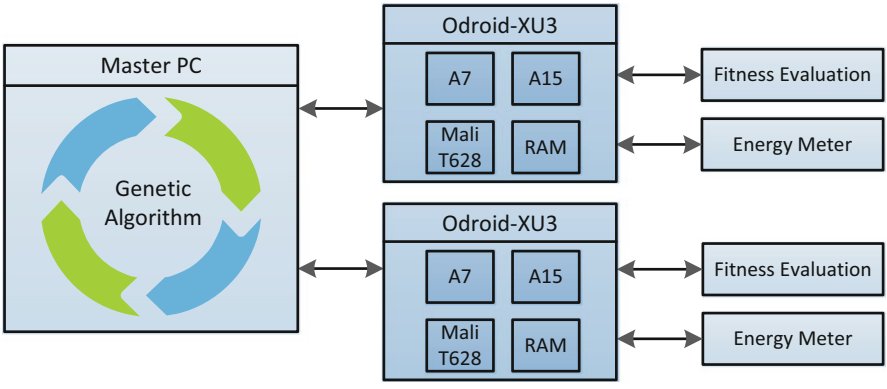


Fig. 26.16 Evolutionary optimization process [33]

In the following, we will focus on parameters dealing with memory configurations. The work group sizes on the Mali-T628 GPU affect the number of threads concurrently running on the GPU and thus have major impact on how fast the data can be processed and how much energy is consumed by the GPU. Partial results within the work group are shared by synchronizing using the shared memory on the streaming multi-processor on the GPU. Thus, memory restricts the parallelism which could be extracted. In addition, the memory allocation size for some of the buffers on the GPU can affect the detection quality. Within the different pipeline steps, ring buffers on the GPU store some of the previously processed images. Depending on the ring buffer sizes, the number of available images varies, e.g., for noise reduction or the feature extraction, which increases/decreases the quality of the results.

To give a detailed example on memory (buffer) allocation optimization, we will now focus on the application of the sensor model and temporal noise reduction applied to the captured images to identify possible virus pixels. According to the PAMONO sensor model, a captured image consists of a background signal, multiplied with a virus signal and an additive noise term. A potential virus pixel can be identified by an increase in intensity. Thus, a sliding window of size b of the past and a sliding window of size a of the future are used to detect potential virus pixels. Since the virus-binding process takes some time, it is not to be seen instantaneously. Thus, a time interval of size g is used to model this attaching process. Figure 26.17 shows the intensity of one pixel over a time series of frames. The detection algorithm calculates the median (red horizontal line) over b and a images. If the difference between the two medians exceeds a specific threshold, this pixel is considered

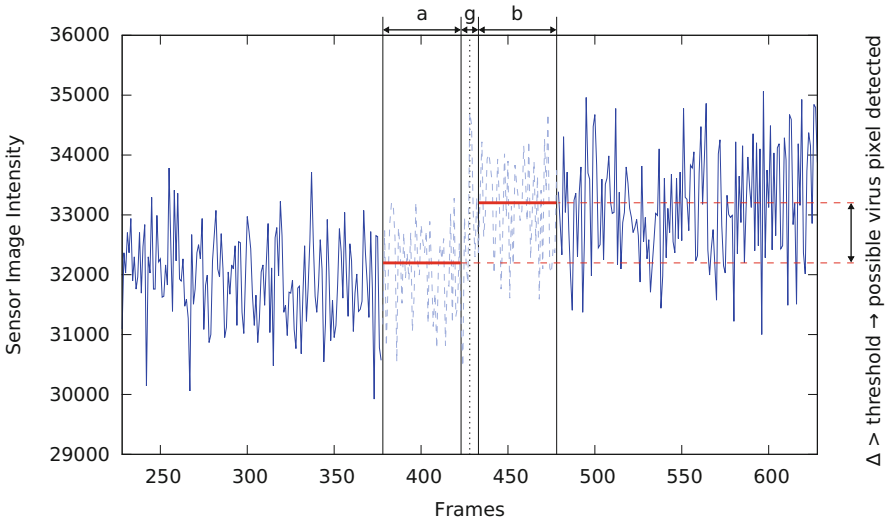


Fig. 26.17 Sensor model application and temporal noise reduction for one pixel and one time step to detect possible virus pixels

as a possible virus pixel. As time moves forward, this is comparable to a sliding window moving over large data. For this preprocessing step, $a + b + g$ images need to be stored in GPU's memory and intuitively, increasing the intervals increases the performance of this processing step. However, the memory is limited, and determining a good combination regarding the other memory parameters in the pipeline is complex. Thus, the genetic algorithm takes care of this selection process. In later pipeline stages, all possible virus pixels are analyzed in more detail, e.g., additional per-pixels and polygon features are extracted.

For the evaluation, we used two data sets for the virus detection program. We used a training and a testing data set, each consisting of 1,000 16-bit gray-scale sensor images with size 706 pixels × 167 pixels. Both data sets were labeled, thus the correct positions of all virus pixels are known, and we can calculate the F_1 score according to Equation (26.31). We conducted three different experiments. Firstly, only hardware parameters were optimized. Secondly, only software parameters were optimized. Thirdly, hardware and software parameters were optimized simultaneously. The unoptimized detection software achieves 7.5 frames per second while reaching the best detection quality. The greatest improvements could be observed for the combined optimization. Figure 26.18 shows the result of this combined optimization experiment. For execution time and energy consumption,

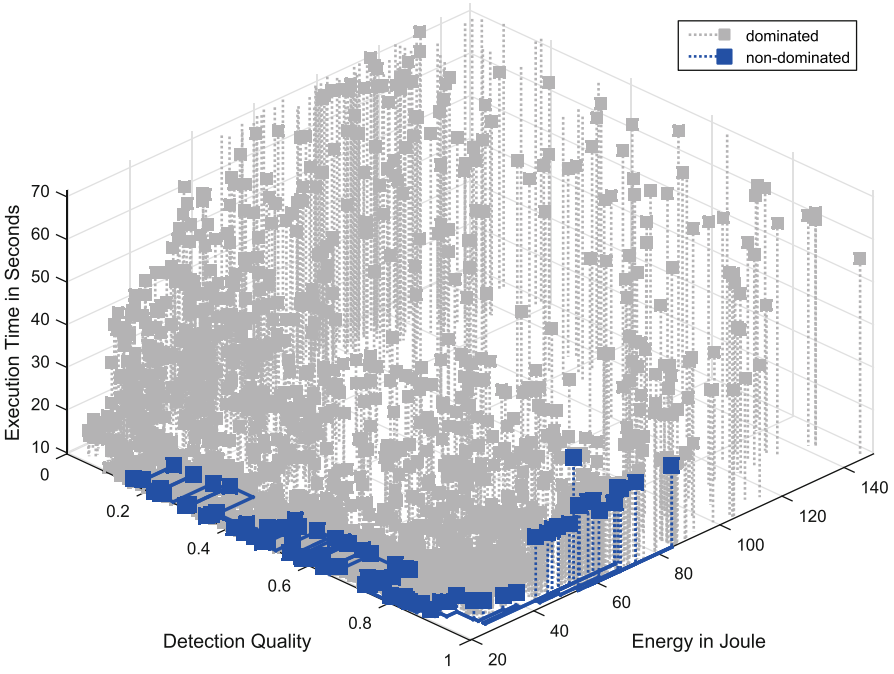


Fig. 26.18 Trade-off between detection quality, energy consumption, and run time resulting from an optimization of hardware and image pipeline parameters. For execution time and energy consumption, lower values are better, and for detection quality, higher values are better [33]

Table 26.1 Excerpt of the Pareto front for the objectives virus detection quality (F_1 training), energy consumption, and execution time. In addition, the detection quality (F_1 testing) for the unseen testing data set is shown. As baseline/comparative measurement, an unoptimized run is given in the first row, which was measured with an unmodified system and program [33]

F_1 training	F_1 test	Energy cons.	Energy sav.	Exec. time	Speedup	Frame rate
100% (fixed)	99.5% (fixed)	370.0 Joule	–	119.8 s	–	7.5 fps
100%	99.5%	57.5 Joule	84%	29.3 s	4.1	30.7 fps
100%	99.5%	84.5 Joule	77%	28.9 s	4.1	31.1 fps
98.5%	97.4%	47.9 Joule	87%	25.5 s	4.7	35.3 fps
97.4%	99.5%	69.3 Joule	81%	23.9 s	5.0	37.7 fps
96.9%	87.8%	27.7 Joule	93%	14.8 s	8.1	60.8 fps
87.9%	76.6%	22.3 Joule	94%	10.8 s	11.1	83.3 fps
84.2%	60.5%	20.7 Joule	94%	11.4 s	10.5	78.9 fps
74.2%	63.9%	23.5 Joule	94%	10.7 s	11.2	84.1 fps
74.2%	64.7%	33.6 Joule	91%	10.4 s	11.5	86.5 fps
51.9%	55.8%	33.0 Joule	91%	10.0 s	12.0	90.0 fps

lower values are better, and for detection quality, higher values are better. The Pareto front is highlighted, and Table 26.1 shows an excerpt of it. Without losing quality, for example, a solution running at 30.7 fps with 84% energy savings was generated. A not 100% detection quality might be sufficient to prove that a sample is contaminated with viruses. By accepting loss of quality, even higher frame rates and energy savings were observed. For example, a solution which still achieves a good detection quality like 76.6% results in an energy saving of 94% and a speedup of more than 11. This indicates that one could use a less capable and thus cheaper hardware or increase the resolution of the camera sensor. An increased resolution enables the simultaneous detection of different virus types. Here, the gold layer is partitioned with different antibodies, and an increased resolution is necessary to detect viruses.

Flexibility with respect to the detection quality is the new objective in this example. This example demonstrates that, in the future, we should not just optimize the usage of the memory in isolation. Rather, it should be included in an overall optimization process for several objectives. This optimization needs to include both the software compilation process as well as the exploration of hardware parameters. This example also demonstrates that hardware parameters exist even in the case of off-the-shelf hardware.

26.7 Conclusions and Future Work

In this chapter, we have demonstrated consequences of the fact that the size of memories has a large impact on their access times and energy consumption. This impact leads to heterogeneous memory architectures comprising a mixture of fast small memories and relatively slow larger memories. Other consequences are resulting from the fact that information processing in Cyber-Physical Systems

has to take a number of objectives and constraints into account. This leads to the idea of an exploitation of memory characteristics such that constraints are met and objectives are used for optimizations. In this chapter, we have presented results of our research groups. First results concern the optimized use of Scratchpad memories for a reduction of the energy consumption. Detailed results are presented for the case of hard real-time systems: we present compiler optimizations using the Worst-Case Execution Times as their objective. We are also briefly describing the optimized mapping to multi-core platforms with a choice of objectives to optimize for. Finally, we demonstrate the integration of memory optimizations into a global approach for the optimization of a cyber-physical sensor system with soft deadlines. In this case, the scope for optimizations comprises software and hardware parameters. The goal is to find good trade-offs between multiple objectives, including quality of service.

We believe that this chapter demonstrates trends in the design of embedded and CPS very nicely. There is a trend from the consideration of just the memory system for mono-processors and single objectives towards whole system optimization for multi-core systems for multiple objectives. The inclusion of the Quality of Service as an objective offers new opportunities, since we can trade off the quality of service against other objectives.

Acknowledgments Part of the work on this section has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis," project B2. URL: http://sfb876.tu-dortmund.de

References

1. AbsInt Angewandte Informatik GmbH (2016) aiT: Worst-Case Execution Time Analyzers. http://www.absint.com/ait
2. AbsInt Angewandte Informatik GmbH (2016) Stack overflow is a thing of the past. https://www.absint.com/stackanalyzer/index.htm
3. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. AFIPS spring joint computer conference
4. Bai K, Shrivastava A (2010) Heap data management for limited local memory (LLM) multi-core processors. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 317–325
5. Banakar R, Steinke S, Lee BS, Balakrishnan M, Marwedel P (2002) Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In: Proceedings of the international symposium on hardware-software codesign (CODES), Estes Park (Colorado)
6. Borkar S (2005) Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro 25(6):10–16
7. Burks A, Goldstine H, von Neumann J (1946) Preliminary discussion of the logical design of an electronic computing element. Report to U.S. Army Ordnance Department, reprinted at https://www.cs.princeton.edu/courses/archive/fall10/cos375/Burks.pdf
8. Chang DW, Lin IC, Chien YS, Lin CL, Su AWY, Young CP (2014) CASA: contention-aware scratchpad memory allocation for online hybrid on-chip memory management. IEEE Trans Comput-Aided Des Integr Circuits Syst 33(12):1806–1817. doi:10.1109/TCAD.2014.2363385
9. Cho H, Egger B, Lee J, Shin H (2007) Dynamic data scratchpad memory management for a memory subsystem with an MMU. In: Proceedings of the conference on languages, compilers, and tools for embedded systems (LCTES). ACM, New York, pp 195–206. doi:10.1145/1254766.1254804

10. Cordes D, Engel M, Neugebauer O, Marwedel P (2013) Automatic extraction of pipeline parallelism for embedded heterogeneous multi-core platforms. In: Proceedings of the international conference on compilers, architectures, and synthesis for embedded systems (CASES), Montreal
11. Ding H, Liang Y, Mitra T (2012) WCET-centric partial instruction cache locking. In: Proceedings of the design automation conference (DAC), San Francisco
12. Dominguez A, Udayakumaran S, Barua R (2005) Heap data allocation to scratch-pad memory in embedded systems. J Embed Comput 1(4):521–540
13. Drepper U (2007) What every programmer should know about memory. http://www.akkadia.org/drepper/cpumemory.pdf
14. Falk H, Kleinsorge JC (2009) Optimal static WCET-aware scratchpad allocation of program code. In: Proceedings of the design automation conference (DAC), San Francisco, pp 732–737
15. Falk H, Lokuciejewski P (2010) A compiler framework for the reduction of worst-case execution times. Int J Time Crit Comput Syst (Real Time Syst) 46(2):251–300
16. Falk H, Verma M (2004) Combined data partitioning and loop nest splitting for energy consumption minimization. In: Proceedings of the international workshop on software and compilers for embedded systems (SCOPES), Amsterdam, pp 137–151
17. Hardkernel Co., Ltd., Odroid-XU3. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127 (2015)
18. Hofmann M, Jost S (2003) Static prediction of heap space usage for first-order functional programs. In: Proceedings of the symposium on principles of programming languages (POPL). ACM, New York, pp 185–197. doi:10.1145/604131.604148
19. HP Labs, CACTI – an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. http://www.hpl.hp.com/research/cacti/ (2015)
20. Jovanovic O, Kneuper N, Marwedel P, Engel M (2012) ILP-based memory-aware mapping optimization for MPSoCs. In: Proceedings of the conference on embedded and ubiquitous computing (EUC), Paphos, Cyprus
21. Kang S, Dean AG (2012) Leveraging both data cache and scratchpad memory through synergetic data allocation. In: Proceedings of the real time and embedded technology and applications symposium (RTAS). IEEE Computer Society, Washington, DC, pp 119–128. doi:10.1109/RTAS.2012.22
22. Kannan A, Shrivastava A, Pabalkar A, Lee JE (2009) A software solution for dynamic stack management on scratch pad memory. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC), pp 612–617
23. Kotthaus H, Korb I, Marwedel P (2015) Performance analysis for parallel R programs: towards efficient resource utilization. Technical Report 1/2015, TU Dortmund, CS Department
24. Li L, Wu H, Feng H, Xue J (2007) Towards data tiling for whole programs in scratchpad memory allocation. In: Proceedings of the Asia-Pacific conference on advances in computer systems architecture (ACSAC). Springer, Berlin/Heidelberg, pp 63–74. doi:10.1007/978-3-540-74309-5_8
25. Liu Y, Zhang W (2015) Scratchpad memory architectures and allocation algorithms for hard real-time multicore processors. J Comput Sci Eng 9:51–72
26. Lokuciejewski P, Cordes D, Falk H, Marwedel P (2009) A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: Proceedings of the international symposium on code generation and optimization (CGO), Seattle, pp 136–146
27. Luican II, Zhu H, Balasa F (2006) Formal model of data reuse analysis for hierarchical memory organizations. In: Proceedings of the international conference on computer-aided design (ICCAD). ACM, New York, pp 595–600. doi:10.1145/1233501.1233623
28. Luke S, Panait L, Balan G, Paus S, Skolicki Z, Popovici E, Sullivan K, Harrison J, Bassett J, Hubley R (2015) ECJ: a java-based evolutionary computation research system. http://cs.gmu.edu/~eclab/projects/ecj/
29. Marwedel P (2010) Embedded system design – embedded systems foundations of cyber-physical systems. Springer, New York

30. McIlroy R, Dickman P, Sventek J (2008) Efficient dynamic heap allocation of scratch-pad memory. In: Proceedings of the international symposium on memory management, pp 31–40
31. National Science Foundation (2013) Cyber-physical systems (CPS). http://www.nsf.gov/pubs/2013/nsf13502/nsf13502.htm
32. Neugebauer O, Libuschewski P (2015) Odroid energy measurement software. http://sfb876.tu-dortmund.de/auto?self=Software
33. Neugebauer O, Libuschewski P, Engel M, Mueller H, Marwedel P (2015) Plasmon-based virus detection on heterogeneous embedded systems. In: Proceedings of the international workshop on software and compilers for embedded systems (SCOPES)
34. Plazar S, Falk H, Kleinsorge JC, Marwedel P (2012) WCET-aware static locking of instruction caches. In: Proceedings of the international symposium on code generation and optimization (CGO), San Jose, pp 44–52
35. Plazar S, Lokuciejewski P, Marwedel P (2009) WCET-aware software based cache partitioning for multi-task real-time systems. In: Proceedings of the international workshop on worst-case execution time analysis (WCET), Dublin, pp 78–88
36. Pyka R, Fassbach C, Verma M, Falk H, Marwedel P (2007) Operating system integrated energy aware scratchpad allocation strategies for multi-process applications. In: Proceedings of the international workshop on software and compilers for embedded systems (SCOPES)
37. Schmoll F, Heinig A, Marwedel P, Engel M (2013) Improving the fault resilience of an H.264 decoder using static analysis methods. ACM Trans Embed Comput Syst (TECS) 13(1s):31:1–31:27. doi:10.1145/2536747.2536753
38. Steinke S, Wehmeyer L, Lee BS, Marwedel P (2002) Assigning program and data objects to scratchpad for energy reduction. In: Proceedings of design, automation and test in Europe (DATE)
39. Suhendra V, Mitra T, Roychoudhury A, et al. (2005) WCET centric data allocation to scratchpad memory. In: Proceedings of the real-time systems symposium (RTSS), Miami, pp 223–232
40. Thiele L, Bacivarov I, Haid W, Huang K (2007) Mapping applications to tiled multiprocessor embedded systems. In: International conference on application of concurrency to system design, pp 29–40. doi:10.1109/ACSD.2007.53
41. Udayakumararan S, Dominguez A, Barua R (2006) Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Trans Embed Comput Syst (TECS) 5:472–511
42. Vera X, Lisper B, Xue J (2003) Data cache locking for higher program predictability. ACM SIGMETRICS Perform Eval Rev 31(1):272–282
43. Vera X, Lisper B, Xue J (2007) Data cache locking for tight timing calculations. ACM Trans Embed Comput Syst (TECS) 7(1):1–38
44. Verma M, Marwedel P (2006) Overlay techniques for scratchpad memories in low power embedded processors. IEEE Trans Very Large Scale Integr Syst 14(8):802–815
45. Wang P, Sun G, Wang T, Xie Y, Cong J (2013) Designing scratchpad memory architecture with emerging STT-RAM memory technologies. In: Proceedings of the international symposium on circuits and systems (ISCAS), pp 1244–1247. doi:10.1109/ISCAS.2013.6572078
46. Wang Z, Gu Z, Yao M, Shao Z (2015) Endurance-aware allocation of data variables on NVM-based scratchpad memory in real-time embedded systems. IEEE Trans Comput-Aided Des Integr Circuits Syst 34(10):1600–1612. doi:10.1109/TCAD.2015.2422846
47. WCET-aware Compilation (2016) http://www.tuhh.de/es/esd/research/wcc
48. Zhang W, Ding Y (2013) Hybrid SPM-cache architectures to achieve high time predictability and performance. In: Proceedings of the conference on application-specific systems, architectures and processors (ASAP), pp 297–304. doi:10.1109/ASAP.2013.6567593

Microarchitecture-Level SoC Design

27

Young-Hwan Park, Amin Khajeh, Jun Yong Shin, Fadi Kurdahi, Ahmed Eltawil, and Nikil Dutt

Abstract

In this chapter we consider the issues related to integrating microarchitectural IP blocks into complex SoCs while satisfying performance, power, thermal, and reliability constraints. We first review different abstraction levels for SoC design that promote IP reuse, and which enable fast simulation for early functional validation of the SoC platform. Since SoCs must satisfy a multitude of interrelated constraints, we then present high-level power, thermal, and reliability models for predicting these constraints. These constraints are not unrelated and their interactions must be considered, modeled and evaluated. Once constraints are modeled, we must explore the design space trading off performance, power and reliability. Several case studies are presented illustrating how the design space can be explored across layers, and what modifications could be applied at design time and/or runtime to deal with reliability issues that may arise.

Acronyms

AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
ASIC	Application-Specific Integrated Circuit
BER	Bit Error Rate
BLB	Bit Lock Block

Y.-H. Park
Digital Media and Communications R&D Center, Samsung Electronics, Seoul, Korea
e-mail: younghwp@uci.edu

A. Khajeh
Broadcom Corp., San Jose, CA, USA
e-mail: amin.khajeh@broadcom.com

J. Yong Shin • F. Kurdahi (✉) • A. Eltawil • N. Dutt
Center for Embedded and Cyber-Physical Systems, University of California Irvine, Irvine, CA, USA
e-mail: junys@uci.edu; kurdahi@uci.edu; aeltawil@uci.edu; dutt@uci.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_28

CA	Cycle Accurate
CDMA	Code Division Multiple Access
CMOS	Complementary Metal-Oxide-Semiconductor
CMP	Chip Multi-Processor
CPU	Central Processing Unit
DFS	Dynamic Frequency Scaling
DMA	Direct Memory Access
DTA	Dynamic Timing Analysis
DTM	Dynamic Thermal Management
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling
ESL	Electronic System Level
GPIO	General-Purpose Input/Output-pin
IDC	Inquisitive Defect Cache
IP	Intellectual Property
IPB	Intellectual Property Block
ISS	Instruction-Set Simulator
ITRS	International Technology Roadmap for Semiconductors
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MPSoC	Multi-Processor System-on-Chip
MTF	Mean Time to Failure
NMOS	Negative-type Metal-Oxide-Semiconductor
PDF	Probability Density Function
PI	Principal Investigator
PMOS	Positive-type Metal-Oxide-Semiconductor
PSNR	Peak SNR
RAM	Random-Access Memory
RDF	Random Dopant Fluctuations
ROM	Read-Only Memory
RTL	Register Transfer Level
SNR	Signal-to-Noise Ratio
SoC	System-on-Chip
SRAM	Static Random-Access Memory
SSTA	Statistical Static Timing Analysis
T-BCA	Transaction-based Bus Cycle Accurate
TLM	Transaction-Level Model
VFI	Voltage/Frequency Island
VOS	Voltage Over Scaling
WCDMA	Wideband CDMA

Contents

27.1 Introduction

A typical System-on-Chip (SoC) is shown in Fig. 27.1. There are four major complicated heterogeneous components in SoCs, such as *processors* (ARM7 shown in the figure), *custom hardware Intellectual Property Blocks (IPBs)* (memory controller, DMA controller, interrupt/GPIO controller and so forth), *on-chip memories* (RAM and ROM) and *on-chip communication architectures* (AHB and APB bus [8]).

These components have their own role such as processors run embedded software and usually control overall operation, custom hardware IPBs are dedicated to execute particular tasks, memories are storage place for data and instructions to be used, and all of which are connected through a on-chip communication architecture consisting of multiple shared interconnected buses using a specific arbitration scheme for fair sharing of the limited bus bandwidth.

27.1.1 A Typical System-on-Chip Design Flow

In recent years, research in this field has focused on the problem of defining a framework for SoC design that promotes Intellectual Property (IP) reuse, with particular attention paid toward achieving performance goals. Such a framework needs to have clearly defined abstraction levels for capturing the SoC design. The basic idea is to model the system first at a high level of abstraction, and then gradually refine the model to create models with higher levels of detail, until we arrive at the gate-level model (netlist). The SystemC [4] or SpecC [25] methodology

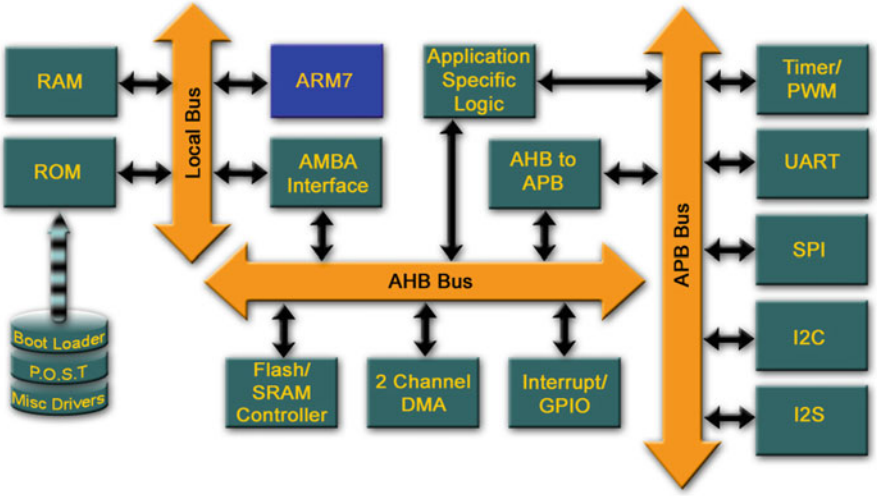


Fig. 27.1 An example of an SoC

focuses on defining a framework in which the system is initially captured at the specification level, and then gradually refined to generate models at lower levels of abstraction. This framework allows reuse of protocol libraries and IPBs at various levels.

Figure 27.2 outlines the typical flow of SoC design in terms of the levels of abstraction at which the designer can simulate the performance of an SoC, and perform communication architecture exploration.

The modeling abstraction levels in Fig. 27.2 are typically used for communication space exploration, with the application/algorithm usually captured with high-level languages such as C/C++. In Cycle Accurate (CA) models, the bus architecture and system components (both masters and slaves) are captured at a cycle and signal accurate level. While these models are extremely accurate, they are too time-consuming to model and only provide a moderate increase in speed over Register Transfer Level (RTL) models. Recent research efforts have focused on using concepts found in the domain of Transaction-Level Models (TLMs) to speed up simulation. Transaction-level models are very high-level bit-accurate models of a system, with specifics of the bus protocol replaced by a generic bus (or channel), and where communication takes place when components call read() and write() methods provided by the channel interface. Since detailed timing and signal accuracy are omitted, these models can be simulated quickly but are only useful for early embedded software development and high-level functional validation of the system. Transaction-based Bus Cycle Accurate (T-BCA) models overcome the slow simulation speed of CA models and the low accuracy of TLMs. T-BCA models capture timing and protocol details, but model components at a less detailed behavioral level, which allows rapid system prototyping and considerable

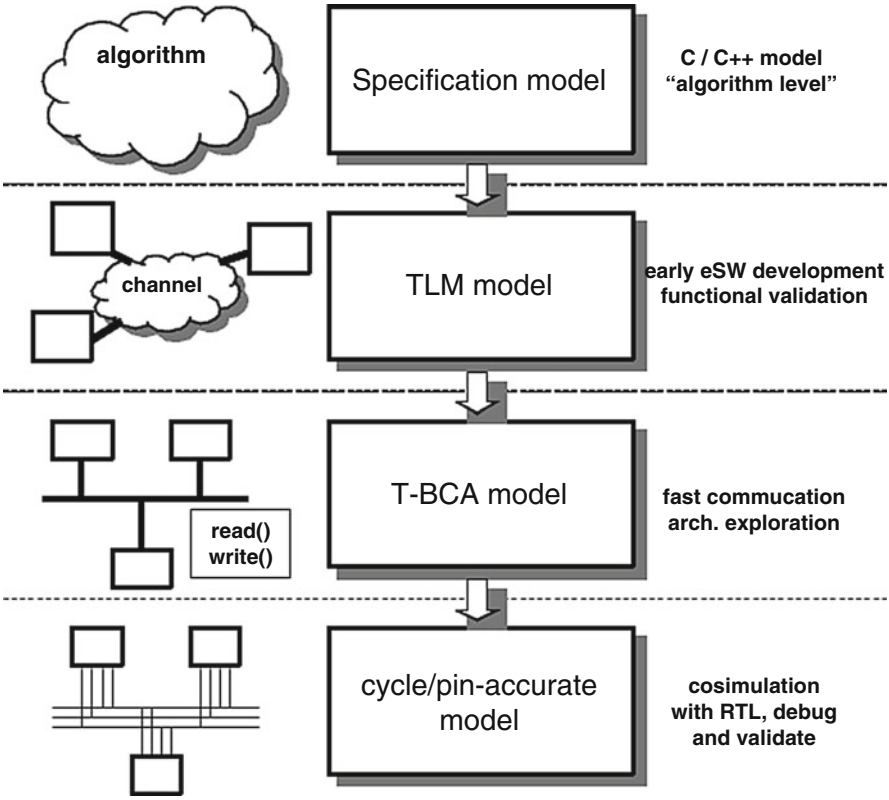


Fig. 27.2 A typical flow for an SoC design [105]

simulation speed over RTL. The component interface and the bus, however, are still modeled at a cycle-accurate level, which enables accurate communication space exploration.

27.2 Power Modeling

Reducing power dissipation is a critical design goal for electrical devices, from handheld systems with limited battery capacity, to large computer workstations that dissipate vast amounts of power and require costly cooling mechanisms. Ever-increasing performance needs, which require large parallel processing at fast clock frequencies accessing huge amounts of data from on and off-chip memories through a very complicated bus interconnection architecture, make this power issue more critical. In reality, their power densities and associated heat generation are exponentially increasing, as shown in Fig. 27.3 [5].This rapid increase of power

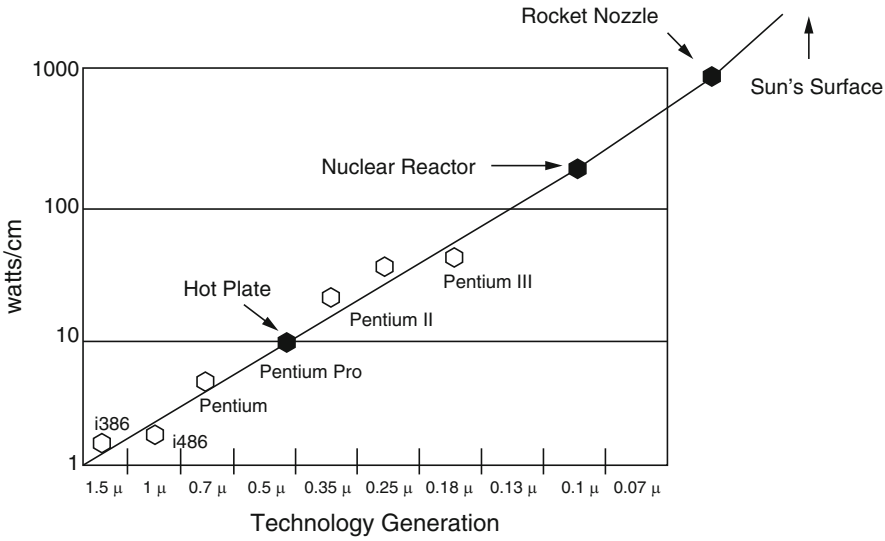


Fig. 27.3 The increase of power densities [5]

dissipation is of great concern, not only because of the aforementioned usage time of handheld devices and cooling costs but also because it can cause many unfortunate side effects, such as damaging chip reliability and reducing expected life cycles.

To address the ever-widening designer productivity gap, TLMs [20,82] and high-level simulation platforms are increasingly being used for SoC architecture analysis and optimization. The increasing importance of power as a design objective in today's complex systems is making it imperative to address power consumption early in the design flow, at the system level, where the benefits of performing power optimizing design changes are the greatest. Since design changes are easier and have the greatest impact on application power dissipation at the system level [58, 109], designers today must evaluate various power optimizations as early as possible in an Electronic System Level (ESL) design flow. In order to explore these optimizations, accurate power estimation models are necessary. These models are especially important for Chip Multi-Processor (CMP) systems with tens to hundreds of processors integrated on a single chip. Even a slight inaccuracy in power estimation for a single processor can result in a large absolute error for the chip. Several system-level power estimation approaches have been proposed in recent years, focusing on the various components of CMP designs, such as processors [58], memories [42], interconnection fabrics [84], and custom ASIC blocks [80]. Because of the heterogeneity of these components, power estimation models are usually customized for each component to achieve desired estimation accuracy. In addition, each type of component requires several power estimation models that can be incorporated at the most coarse grain, high levels of abstraction, as well as at the most detailed, low-level simulation abstractions.

27.2.1 Sources of Power Consumption and Defining Energy

In digital Complementary Metal-Oxide-Semiconductor (CMOS) circuits, there are three key sources of power consumption, shown below [87]

$$P_{total} = P_{dynamic} + P_{short-circuit} + P_{leakage} \tag{27.1}$$

Decomposed equations for each source, respectively, can be described by the following equations. Firstly, dynamic power consumption can be derived as below:

$$P_{dynamic} = \alpha_{0->1} C_L {V_{dd}}^2 f_{clk} \tag{27.2}$$

where $\alpha_{0->1}$ is the probability that a power consuming switching occurs, C_L is the load capacitance, V_{dd} is the supply voltage, and f_{clk} is the clock frequency. Note that Eq. 27.2 shows an important characteristic of dynamic power, which is that the power is quadratically proportional to the supply voltage, and can be efficiently reduced as the supply voltage level is reduced.

Secondly, the short circuit power consumption is formulated as below:

$$P_{short-circuit} = I_{short-circuit} V_{dd} \tag{27.3}$$

where V_{dd} is the supply voltage, and $I_{short-circuit}$ is the short circuit current which arises when both the NMOS and PMOS transistors are concurrently turned on, making a direct path from the supply power to ground. However, since this short circuit power consumption is responsible for only 10–15% of total power consumption and researchers have not found a good way to reduce this power without sacrificing performance [87], we will not focus on this component of power consumption in detail.

Finally, leakage power consumption can be calculated by:

$$P_{leakage} = I_{leakage} V_{dd} \tag{27.4}$$

where $I_{leakage}$ is the leakage current and V_{dd} is the supply voltage. Besides the dynamic and short-circuit power, transistors also consume leakage power (also referred to as static power), which is quickly becoming the large portion of total power consumption, based on the recent International Technology Roadmap for Semiconductors (ITRS) 2008 update [5], as shown in Fig. 27.4. Unlike dynamic power, the leakage power consumption continues during logic's idle status, and most of the techniques for dynamic power saving are not helpful for leakage power saving [5]. There are two major sources of these leakage currents, which are subthreshold leakage and gate-oxide leakage.

The first major component of the leakage current is gate-oxide leakage current, which flows from the gate of a transistor into its substrate. The thickness of the oxide

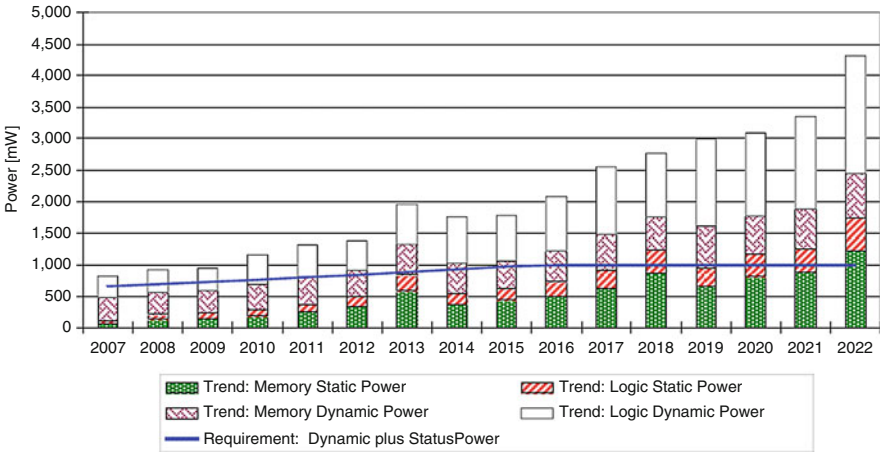


Fig. 27.4 Power consumption trends of portable chips [5]

material that insulates the gate decides this leakage current. The equation [87] for this type of leakage is given by:

$$I_{gate-oxide} = K_1 W \left(\frac{V_{dd}}{T_{ox}}\right)^2 e^{-\alpha \frac{T_{ox}}{V_{dd}}} \tag{27.5}$$

where W is the gate width, T_{ox} is the thickness of the oxide, V_{dd} is the supply voltage, and K_1 and α are constants. Based on Eq. 27.5, the gate-oxide leakage, $I_{gate-oxide}$ will be increased exponentially by decreasing the thickness (T_{ox}) of the oxide material of the gate. In conjunction with other design parameters, such as transistor sizes and supply voltage, the thickness of the transistor (including T_{ox}) for the upcoming chip design will also be decreased, and this will cause the exponential increase of the gate-oxide leakage current. Insulating the gate with high-k dielectric material might be the best possible solution for the problem of increasing gate-oxide leakage for the next few years [5].

The second major component of the leakage current is subthreshold leakage current, which flows between the drain and source terminal of a transistor. When the gate-source voltage, V_{gs}, exceeds the weak inversion point, it is still lower than the threshold voltage, V_t, and the MOSFET works like a bipolar transistor. The subthreshold current in this region changes exponentially, depending on the gate-source voltage, V_{gs}. The current in this subthreshold region is formulated by [5]:

$$I_{subthreshold} = K_2 W e^{\frac{-V_t}{nT}} \left(1 - e^{\frac{-V_{dd}}{T}}\right) \tag{27.6}$$

where W is the gate width, V_t is the threshold voltage, V_{dd} is the supply voltage, T is the temperature, and K_2 and n are constants. According to the Eq. 27.6, when the threshold voltage, V_t, is reduced, the subthreshold leakage current is increased

exponentially. In response to increasing requirements for reducing technology scale parameters for upcoming chip designs, the threshold voltage should be reduced in conjunction with the supply voltage, and this causes a worse problem of subthreshold leakage. The increased subthreshold leakage current can cause another serious problem, called thermal runaway. A vicious cycle can result, in which the increased leakage currents cause increased temperature, then the increase temperature again causes more leakage currents, based on Eq. 27.6.

On the other hand, energy can be defined by the total quantity of the work a system completes over a period of time and formulated as the following equation :

$$E(\mathscr{T}) = \int_0^{\mathscr{T}} P(t)dt \tag{27.7}$$

where E is energy, $P(t)$ is the instantaneous power at time t, and T is a time interval. The unit for energy is joules (J), and the unit for power is watts (W). In a computing system, power is the rate at which the system consumes the supplied electricity while performing computing activities, and energy is the total amount of consumed electricity over time for the task [87].

Note that some techniques to decrease power do not always decrease energy consumption. For instance, the power used by a computing system can be reduced to half by halving the supplied clock frequency; however, the total energy consumed will be approximately the same, because computing time will be twice as long to run the same task.

Different approaches will be necessary for reducing power or for reducing energy, depending on the context. For a system (e.g., a workstation) in which temperature is an important concern (because high temperature can cause various problems such as decreasing the overall speed of chips, increasing cooling costs, damaging chip reliability, and causing the thermal runaway problem), we must reduce instant power, despite the influence on overall energy, to keep the temperature of the system within tolerable limits. In handheld systems, however, reducing energy is usually the more important issue, because it is directly related to the battery lifetime.

27.2.2 Overview of Power Saving Techniques

There are many proposed techniques to reduce the power and energy of digital systems. The discussion in this section provides an overview for the most widely used power saving techniques.

Approaches for minimizing power consumption in CMOS digital systems involve various design abstraction levels, from the software algorithm and architectures to circuits. Some important power saving techniques are summarized in Table 27.1. They are classified by enabling time and sources of targeted power consumption (leakage/dynamic). Some techniques can be employed at the time of

Table 27.1 Summary of power saving techniques

Power	Design time	Idle time	Run time
Dynamic	Lower Vdd Multi-Vdd Transistor sizing Logic optimizations	Clock gating Operand isolation	Dynamic voltage Scaling (DVS) Dynamic frequency Scaling (DFS)
Leakage	Multi-Vt	Sleep transistors Multi-Vdd Variable Vt	Variable Vt

design, such as modification of transistor size and logic optimization, while other techniques, including varying supply voltage, clock frequency, and threshold voltage, can be either implemented statically at the design time or applied dynamically during the run time [84].

There are several techniques to decrease dynamic power consumption in particular. These techniques have different trade-offs, and some of them do not necessarily reduce the total energy consumption.

Clock gating (CG) is a widely used power optimization technique that saves dynamic power by stopping clock supply to unused portions in synchronous logic designs. "*Multistage clock gating*" refers to the scenario in which a clock gating cell controls either another one, or an entire row of clock gating cells. The synthesis tool identifies common *enables* and groups them with another clock gating cell. This technique is used for further optimization of existing gating cells by merging more register banks, so that the clock gating can be moved up closer to the root (i.e., the power/ground pad(s)), for more power savings [2].

Operand isolation is the technique that keeps the inputs of the data-path operators stable whenever the output is not used. Special circuitry is required to identify redundant computations of data-path components and to prevent unnecessary switching activity. Both CG and OI techniques can be implemented automatically with standard tools such as Synopsys Power Compiler [2], or manually, by inserting necessary circuits at the RTL.

Reducing the physical load capacitance is a technique to reduce dynamic power consumption. Low-level design parameters, such as size and wire length of transistors, decide this physical capacitance. We can reduce this capacitance by decreasing transistor sizes or by decreasing wire length and/or width at the cost of performance degradation.

Dynamic Frequency Scaling (DFS) is a technique which varies the clock frequency during run time. This technique can reduce dynamic power consumption linearly (Eq. 27.2). However, this also degrades overall performance and does not save total energy consumption. Thus, we can use this technique when reducing peak or average power dissipation, when reducing the temperature of the chip is the major concern.

Lowering the supply voltage is a very attractive method of power saving, because it reduces dynamic power quadratically (Eq. 27.2), while reducing leakage

power (Eq. 27.4). However, this technique also increases the delay of CMOS gates inversely. Thus, logical and architectural compensation is necessary for this degradation of performance. The technique of scaling the supply voltage during run time is called *Dynamic Voltage Scaling (DVS)*. However, since reducing the voltage increases gate delays, we also have to reduce the clock frequency for proper operation of the circuit. *DVS* is therefore commonly used in conjunction with *DFS*.

On the other hand, there are techniques that primarily decrease the subthreshold leakage power.

Multiple threshold voltages (V_t) provide a trade-off for leakage power and speed. The high-V_t transistor has a leakage current that is roughly one order of magnitude lower than that of the low-V_t transistor, at the cost of reduction in performance. Thus, the low-V_t transistors are preferred for use in timing critical paths, whereas the high-V_t transistors are used for the rest of the paths. According to Eq. 27.6, this technique exponentially decreases the subthreshold leakage. However, increasing the threshold voltage can decrease logic performance as well, as described in the equation below :

$$f \propto \frac{(V_{dd} - V_t)^{\alpha}}{V_{dd}} \tag{27.8}$$

where f is a frequency, V_{dd} is the supply voltage, V_t is the threshold voltage, and α is a constant.

Decreasing the size of circuits can reduce leakage power. This is because the total leakage current is proportional to the leakages that are consumed in all transistors in a circuit. Minimizing cache size and reducing unnecessary logic in the chip will be helpful in reducing the actual number of transistors and the corresponding leakage power. However, this is not always possible, because reduced logic may degrade performance.

Power gating with *sleep transistors* is reducing the count of the active transistors dynamically, by blocking the power supply to the idle portion of circuits. Problems with this method might include difficulty in predicting the exact time and portion of the idle part of various components, and minimizing the overhead for this by turning them off or on.

Cooling the system is also helpful in reducing leakage power. Various cooling techniques such as blowing cold air, refrigerating the system, or even circulating costly liquid nitrogen have been proposed and used for several decades. This technique has several advantages, such as decreasing subthreshold leakage power significantly and preventing degradation of reliability and lifecycle of a chip. This technique also increases the overall speed of chips, because electricity has smaller resistance at lower temperature. In spite of the aforementioned advantages, cost and cooling system power consumption are major limitations to applying cooling technology to every chip [87].

Again, only the most popular and widely used power minimization techniques have been presented in the section.

27.2.3 Overview of System-Level Power Estimation Methodologies

There have been several power estimation methodologies for specific components in an SoC such as processors [19,27,48,64,89,94,98], various communication fabrics [36, 83–85] and memories [67], and developed power examination tools such as SimplePower [109] and Wattch [19]. There are relatively few methodologies for customized ASIC blocks, due to their extreme heterogeneity. Few researchers have tried to make comprehensive power models for all these components [11, 58, 78]. These models still simplify the power model for a specific component (e.g., the two state processor power model for PowerViP [58]).

With the conventional approach, designers need power estimation models at each of the design abstraction levels, in order to guide design decisions that affect power dissipation. Existing power estimation techniques create power models that map onto, and are useful only at, a particular level. For instance, a technique that is readily applicable at the detailed functional level cannot be easily used at the higher functional level, which is unaware of the detail functionality of the design. Furthermore, if this technique is used at the lower levels, it fails to exploit the additional accuracy in the control and data paths and suffers from an abstraction mismatch. Similarly, cycle-accurate power estimation tools are applicable to the detailed microarchitectural level of the ESL design flow, but cannot be easily ported to higher-level architectural models that lack microarchitectural detail such as the methods described in ▶ Chaps. 25, "Hardware-Aware Compilation" and ▶ 26, "Memory-Aware Optimization of Embedded Software for Multiple Objectives". The mismatch between power model granularity and level of detail captured at an ESL design level thus limits the applicability of current power estimation techniques across an ESL flow.

In [81], a comprehensive multi-granularity power model generation methodology that spans the entire ESL design flow (Fig. 27.5) was reported. Using industry-standard design flows (Fig. 27.6), this methodology can quickly generate multiple power models ranging from the simplest two-level, coarse-grained model for early power estimation, to the most accurate cycle-accurate model (Fig. 27.8) that allows designers to explore the impact of using power optimizations with minimal manual interference and effort. Our proposed approach is based on the concept of hierarchical decomposition. This decomposition is aided by a tripartite hypergraph model of processor power that can be iteratively refined to create power estimation models with better accuracy (Fig. 27.7). The methodology serves a vital function in supplying a designer with multiple derivative processor power estimation models that match the increasing accuracy of the design, as it is successively refined from the functional, to the architectural and then down to the cycle-accurate microarchitectural stages in an ESL design flow (Figs. 27.7 and 27.8). The feasibility of this approach was demonstrated on an OpenRISC and MIPS processor case study, and present results to show how multi-granularity power models generated for the processors provide designers with the flexibility to trade-off estimation accuracy and simulation effort during system-level exploration.

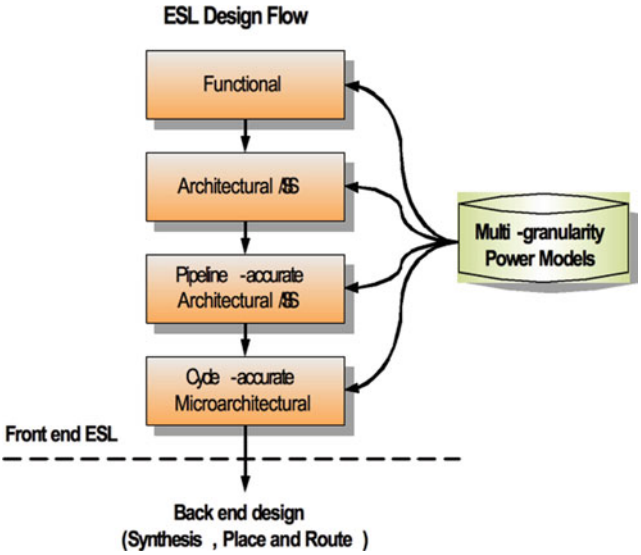


Fig. 27.5 ESL design flow for embedded processors

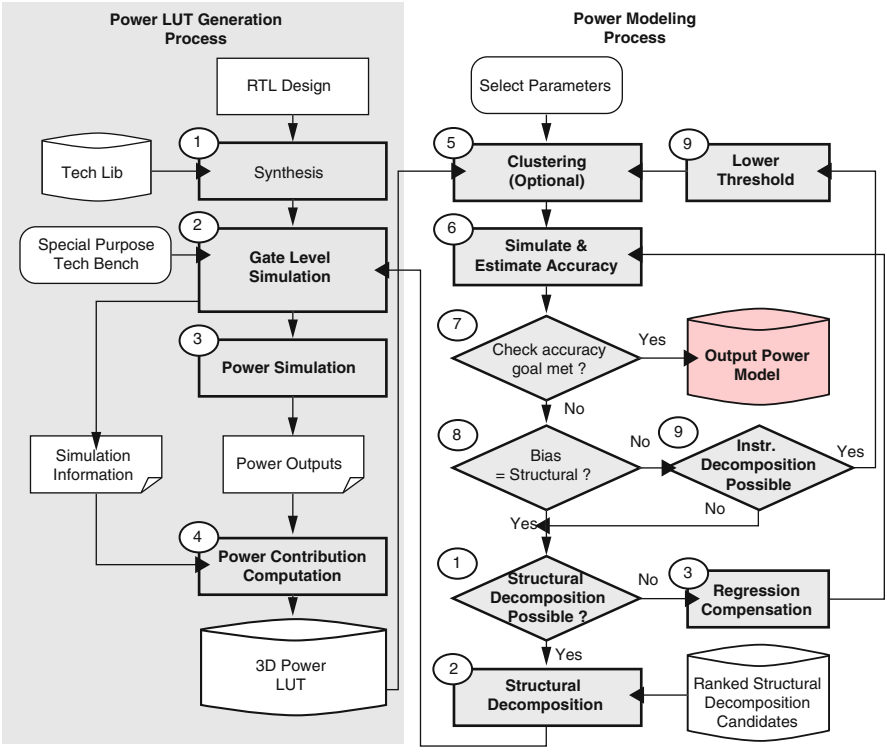


Fig. 27.6 Power model generation methodology

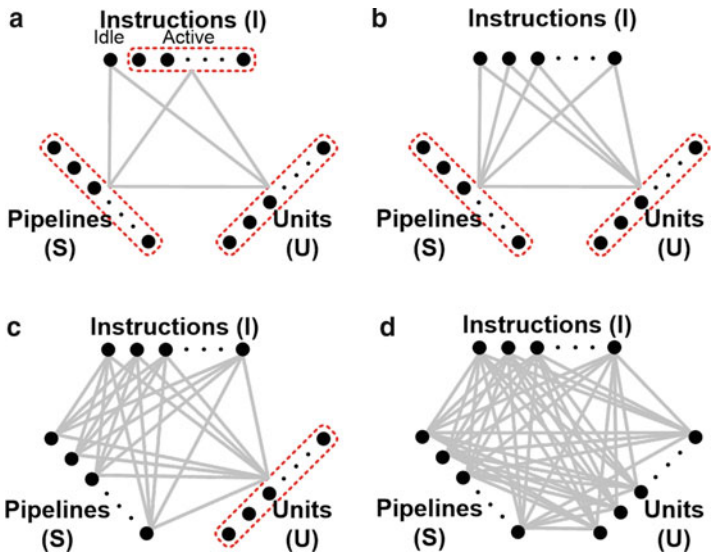


Fig. 27.7 Tripartite hyper-graph H(P), (**a**) simplest two-state power model, (**b**) power model with set I decomposed, (**c**) power model with sets I, S decomposed, (**d**) power model with sets I, S, U decomposed

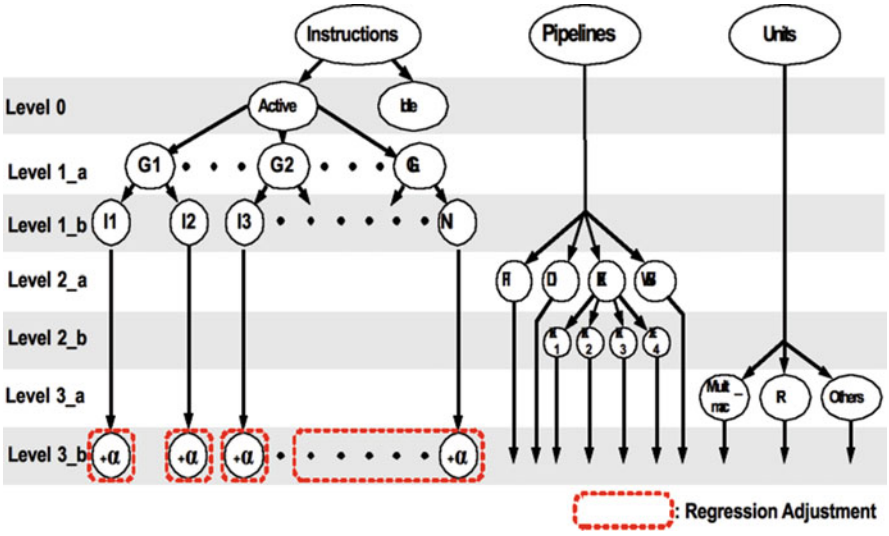


Fig. 27.8 Hierarchical power model for OpenRISC processor

Figure 27.9 shows the average absolute cycle error (E_{AAC}) and relative estimation effort in terms of simulation overhead, for the generated power models for OpenRISC. The power model at Level 0 has a large error of over 20%, which subsequently reduces for the more detailed power models. The Level 3_b power

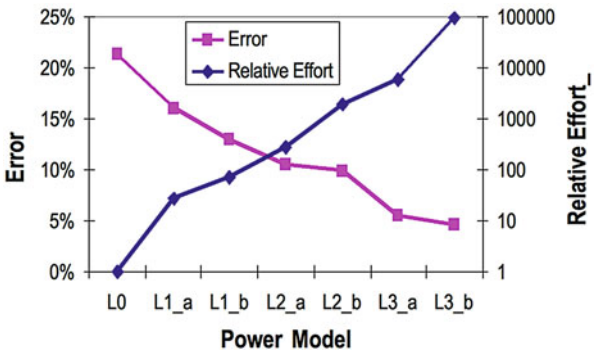


Fig. 27.9 Average absolute cycle error and relative effort for power models

model has an approximately 5% error, which is extremely good compared to gate level estimates. The error in such a detailed model occurs because of several factors, such as the inability to capture the layout and consequently accurately model intra-processor interconnect length, and wire switching.

Figure 27.10 shows a comparison between system-level and gate-level normalized power for the "mul" testbench executing on OpenRISC, across different ESL design flow levels. The figure shows how the coarse-grained *Level 1_b* instruction-set model at the architectural/ISS level is unable to track the power variation very accurately due to the absence of a pipeline at that level. When the pipeline is captured, as in the *Level 2_b* case, then accuracy improves slightly. However, it requires a more detailed *Level 3_b* model which additionally captures the structural units in a cycle-accurate manner, to accurately track the peaks of the gate level power waveform. The power estimated at this level can allow designers to accurately estimate peak power of the processor at simulation speeds that are $100 - 1000\times$ faster than gate-level power simulation. Such a model is extremely useful for determining the thermal and electrical limits of the design and can guide the selection of the appropriate packaging to prevent hotspots and thermal runaway.

27.2.4 Cache Power Modeling

Even though the processor power model described above deeply investigated only the power of core components, we can take account of the cache power for the more realistic embedded processor power model with equation below:

$$P_{processor} = P_{core} + P_{cache} \tag{27.9}$$

where $P_{processor}$ is the cycle-accurate power for the entire processor, P_{core} is cycle-accurate power of the core (which is available from the our methodology), and P_{cache} is the cycle-accurate power for the cache component (which may be obtained from the available memory tool such CACTI [42]).

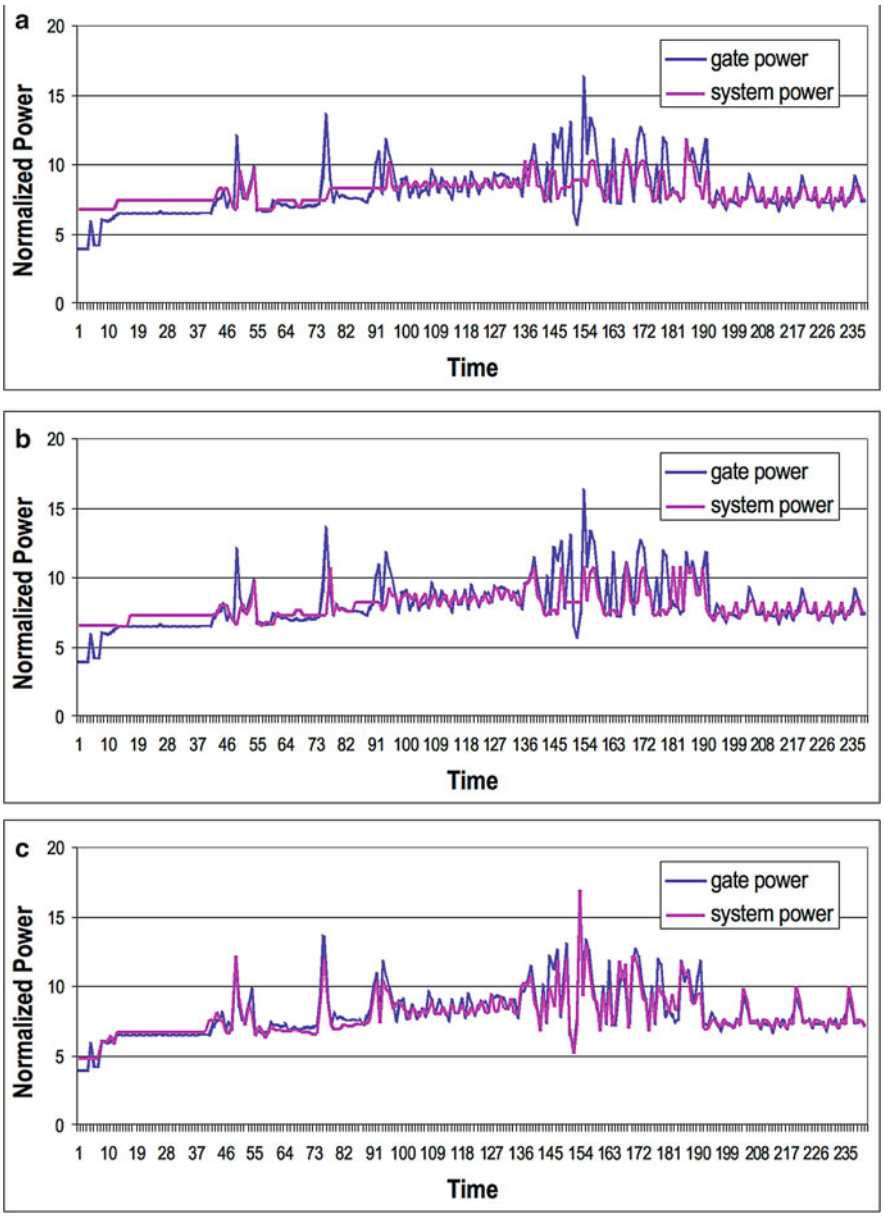


Fig. 27.10 Relative power waveform comparison for the "*mul*" testbench on OpenRISC (Unit for Time: 20 ns). (**a**) Level 1_b. (**b**) Level 2_b. (**c**) Level 3_b

If we are interested in simple average power consumption of a processor, the power model can be formulated as:

$$P_{processor\_avg} = P_{core\_avg} + \frac{nP_{cache\_hit}}{N} + \frac{mP_{cache\_miss}}{N} \tag{27.10}$$

where $P_{processor\_avg}$ is the average power value for the entire processor, P_{core_avg} is the average power value of the core, P_{cache_hit} is the average power value of cache hit (access power), P_{cache_miss} is the average power value of cache miss (idle power), n is total cache hit time during the execution, m is the total cache miss time during the execution, and N is the total execution time.

Rodriguez et al. [92] investigated power consumption of cache with varying sizes (from 16 to 256 K) and associativities (from 1-way to 16-way) for the 65 and 32 nm technology libraries as shown in Fig. 27.11. In fact, current and future SoC designs will be dominated by embedded memory as projected by the ITRS reports which indicate that memories will continue to be a major fraction of any SoC in terms of both area and power [46].

27.3 Thermal and Reliability Issues and Modeling in the Nano-CMOS Era

Downscaling of chips or the continued shrinkage in gate length has naturally increased the power density of chips. Resulting high temperature of chips became one of the biggest issues in chip design, and those thermal issues are becoming more problematic with aggressive technology scaling. In extreme cases, some parts of a chip can be burned out leading to chip failure in the end; thermal runaway, which is caused by positive feedback between increased leakage current and high temperature, can be thought of as one example of such a case. In addition, as we put a lot of heterogeneous components on a chip, the thermal distribution of a chip tend to become nonuniform, i.e., some parts of a chip are hotter than the others due to different processing tasks in different parts of a chip. Implementing multiple cores instead of increasing the clock frequency of a single core became a trend in processor design as a way of alleviating the burden of high power consumption and enormous heat generation [38], and this trend also plays a role in making the thermal distribution nonuniform over a chip to some extent. Especially when thread mapping among the multiple cores is not well-balanced, nonuniform thermal distribution can become a lot worse, resulting in multiple localized temperature maxima, which are usually termed hotspots [38]. According to [14, 15, 103], temperature within a chip can vary as much as 50 ℃ across a die, and examples of this nonuniform thermal distribution are given in Fig. 27.12.

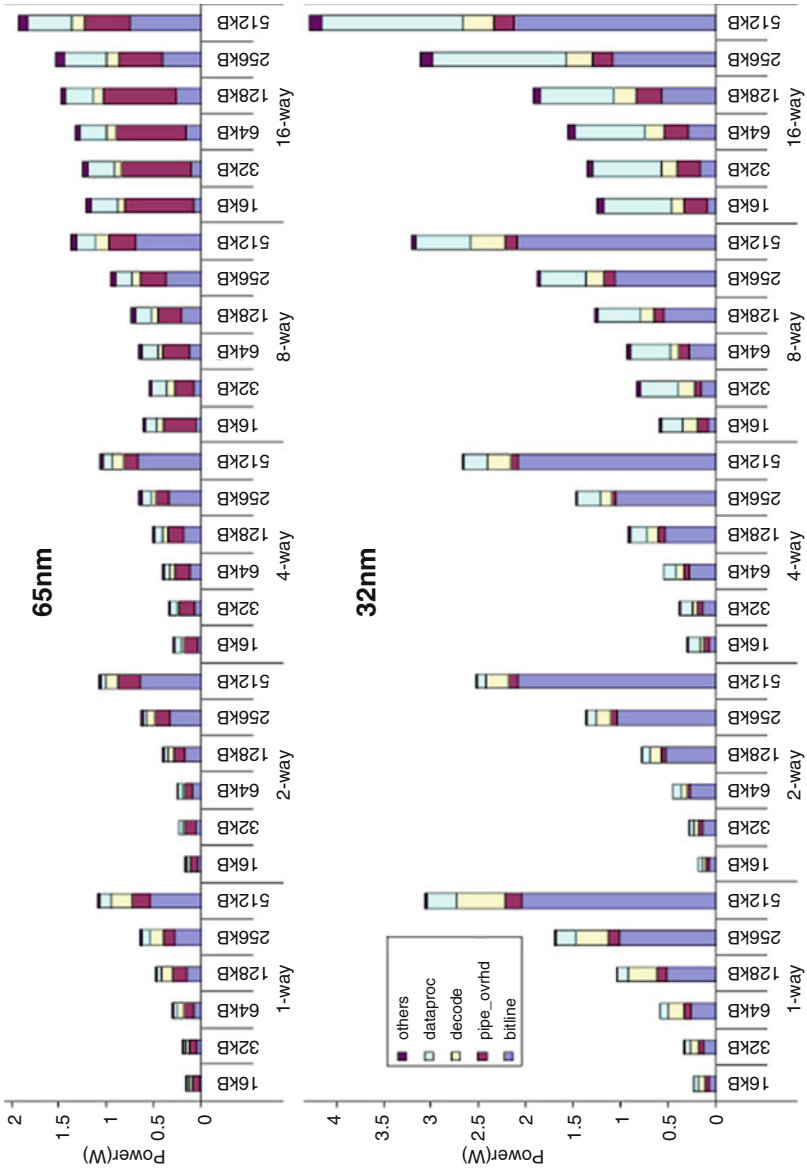


Fig. 27.11 Power breakdown for the 65 nm and 32 nm libraries varying cache sizes and associativities [106]

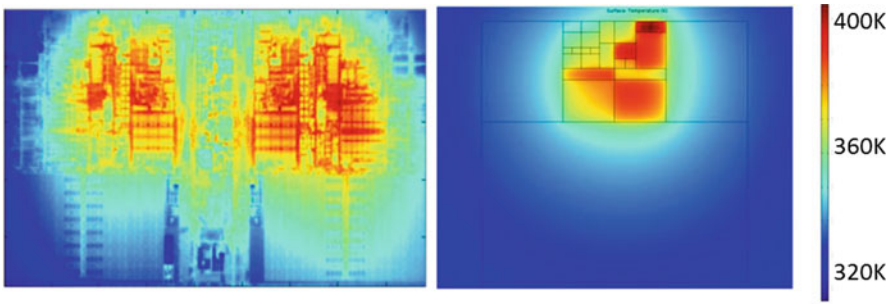


Fig. 27.12 Examples of nonuniform thermal profiles [77, 111]

Hotspots and thermal gradient may result in various kinds of issues: reduced reliability of a chip due to electromigration, [14] timing failure or communication error between functional blocks in a chip due to increased clock skews, higher cost than before for cooling solutions such as heavy cooling fans, heat sinks, etc. [103]

27.3.1 Reliability

One of the serious issues that can be caused by high operating temperatures and a nonuniform thermal distribution over a die is the reduction in the reliability of interconnects and the resulting short life expectancy of a chip due to electromigration [14]. Electromigration is the result of momentum transfer from the collision between electrons and the atoms forming the lattice of the material, and it can cause void or hillock formation along the metal lines in extreme cases. With CMOS technology scaling, the reliability and the life expectancy of interconnects in a chip are becoming more susceptible to electromigration than before. Black's equation or its modified equation [15] given below have been widely used as a way of modeling and predicting the Mean Time to Failure (MTF) of interconnects subjected to electromigration:

$$MTF = \frac{A}{J^n} e^{\frac{E}{kT}} \tag{27.11}$$

In this equation, A is a constant that is determined by the material properties and the geometry of the interconnects, J is the current density, n is a scaling factor that is to be determined experimentally, E is the thermal activation energy depending on the used material, k is the Boltzmann's constant, and T is the absolute temperature of the metal in the unit of kelvin. The current density exponent n is usually set to a value between one and two, and it depends on the failure mechanism [79]; a value close to one characterizes well the failure due to void growth [62]; a value close to two represents the failure due to void nucleation quite well [100]. In the equation, two dominant factors determining the MTF of interconnects are the current density

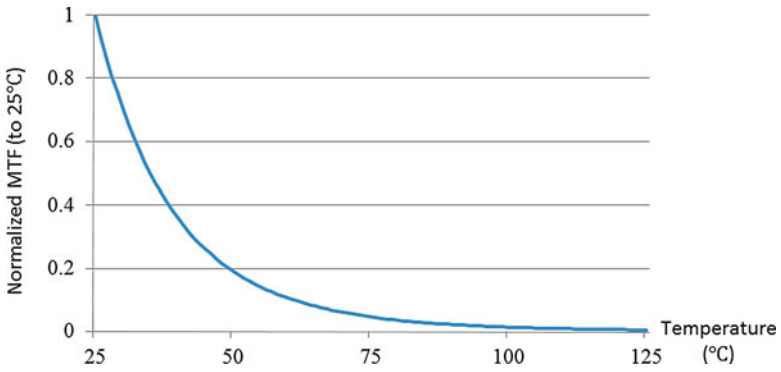


Fig. 27.13 Trend in MTF as a function of temperature

J and the temperature *T*. As CMOS technology scales down, the current density of interconnects generally increases [60], so the life expectancy of interconnects will decrease. To make it worse, the MTF decreases exponentially with respect to the temperature of interconnects. For example, when the temperature of an interconnect changes from 45 to 65 ℃, the life expectancy of the interconnect is reduced by 70% roughly, and the chip will fail much sooner than before if we design chips in a traditional way without proper consideration on thermal issues and adequate cooling solutions. The trend in MTF, which is normalized so that the MTF at 25 ℃ is to be one, is given in Fig. 27.13 as a function of temperature.

As process scaling develops further, the top metal layers get closer to the substrates, and this will further intensify the impact of thermal gradients of substrates on the thermal profile of interconnects [45]; thus, the reliability or the MTF of interconnects decreases exponentially with the increase in the temperature of substrates. In order to improve the reliability or the MTF of interconnects, it becomes indispensable to manage the thermal distribution of a chip dynamically and also to consider the thermal distribution of substrates during chip design or interconnect design stage so that we can avoid hot regions or hotspots on the substrates for the routing.

27.3.2 Dynamic Thermal Management

As we discussed in previous sections, temperature plays a critical role in the reliability, the performance, and the power consumption of a chip in current and future CMOS technology nodes. Therefore, temperature of a chip, especially in case of a high performance chip, should be managed in a smart way at run time so that the maximum temperature can be controlled and also temperature can be evenly distributed both temporally and spatially for better reliability and performance of a chip. According to [39], cost for the implementation of cooling and packaging solutions was expected to increase at an alarming rate with the thermal dissipation of

65 W or higher; hence, thermal management of a high performance chip is also quite crucial in terms of cooling and packaging cost. A large number of techniques for Dynamic Thermal Management (DTM) have been proposed and developed in recent years as ways of limiting the peak temperature of a chip or managing the temporal and spatial temperature variation of a chip through proper resource management [18, 53]. Those techniques can be roughly classified into one of two categories based on how the source management is performed: hardware-based techniques and software-based techniques.

Hardware-based DTM

The relationship between temperature and power dissipation is quite complicated, but temperature can be managed to a certain extent by controlling power consumption of a chip. One of the simple power management techniques, which is called clock gating, began to be used generally in the early 2000s [39]; dynamic power consumption can be minimized by disabling the clocks in a functional block when the functional block is not in use or when the temperature of the functional block reaches a threshold. Clock gating is relatively simple to implement and has good cooling capability because we can effectively reduce the power consumption of a clock tree, which may consume up to around 70% of total dynamic power [37], but the performance degradation is quite high.

Changing dynamically the supply voltage and the clock frequency of a processor based on the workload can be effective in reducing the dynamic power consumption because of the quadratic relationship between dynamic power and the supply voltage, and this technique is called Dynamic Voltage and Frequency Scaling (DVFS) [102]. In case of a processor consisting of multiple cores, the supply voltage and the clock frequency settings of each core can be scaled independently, and it is termed local DVFS or distributed DVFS or per-core DVFS [26], while the chip-level voltage and frequency control is usually termed global DVFS [37]. Additional hardware components and increased design complexity to support multiple clock domains or multiple Voltage/Frequency Islands (VFIs) might become a critical issue especially in case of processors with a large number of cores [40].

Fetch gating [10, 102] is another way to cool down a chip through power consumption reduction; it controls the instruction activity in the pipeline by throttling the fetch stage, and its performance on power reduction and thermal management highly depends on the implemented throttling mechanisms as expected.

Software-based DTM

A simple temperature-aware task scheduling technique for single-threaded processors was proposed in [93]; kernel monitors the CPU activity of each process and the temperature readings from a thermal sensor. When the temperature of a chip becomes higher than a threshold, the kernel identifies processes that use more CPU activities than a predefined value, and then slows them down for cooling purpose. Even though it was simple, it worked effectively to some extent. This basic idea was extended to temperature-aware scheduling techniques for processors that support multi-threading or have multiple cores. For example, a temperature-aware

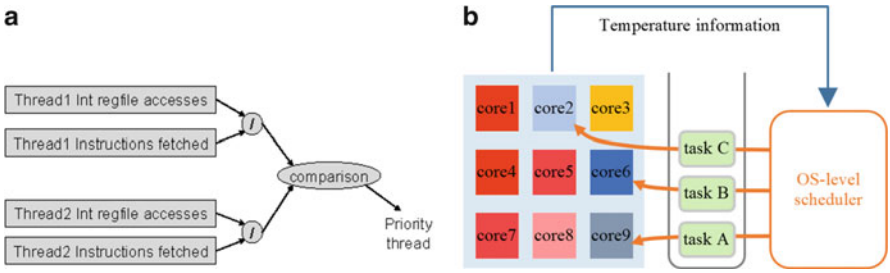


Fig. 27.14 (**a**) Thread selection when the integer register file is thermally critical [32], (**b**) Thermal aware task scheduling for MPSoC [29]

scheduling technique for simultaneously multi-threading (SMT) processors was proposed in [32]; it manages the execution of threads selectively and dynamically based on the probability of heat generation of each thread, and hardware event counters [56] are used for the estimation of the heat generation probability. In [29], a scheduling method specifically targeting Multi-Processor Systems-on-Chips (MPSoCs) was proposed; for each core or processor, the probability of workload assignment is calculated and updated regularly based on the temperature history in the past, and one core with the highest probability is selected when a new workload assignment is required.

When there are multiple cores or processors in a chip, process or task migration can be used effectively in order to balance the thermal distribution among all cores and also to improve the performance as a result; in [33] , a task migration technique was used on top of local DVFS, and it successfully avoided all thermal emergencies, and also achieved 2.6 times speedup when compared with the base case of using local clock gating without task migration. Figure 27.14 illustrates such systems.

27.3.3 Thermal Sensors

As discussed in previous sections, DTM solutions use temperature information to manage the thermal distribution of a chip. Performance Counter-based temperature information can be used for thermal management [56], but the information is not a direct representation of thermal behaviors of a chip most of the time, and it can supply approximation at best. In that sense, it is far better to use the temperature information from thermal sensors because it represents actual thermal behavior of a chip. Each thermal sensor basically provides point-wise temperature information. Thus, it would be better to use a large number of thermal sensors in order to have correct temperature information at any locations of interest on a chip. As for the locations of interest, hotspots need to be monitored first for better reliability and performance, and also for the reduction in power consumption of a chip just as we discussed in previous sections. In addition, a lot more thermal sensors need to be

deployed across a die so that the thermal distribution over a die can be monitored and balanced out for the increased reliability of a chip and also for the prevention of performance degradation. However, it is not reasonable to allocate as many thermal sensors as possible on a small-sized chip in reality due to a lot of practical design constraints [63] power consumption and heat generation of thermal sensors, routing and placement issues, etc. As a result, quite a large number of methods have been proposed regarding how to select the number of thermal sensors properly and how to allocate them on a die in order to have accurate temperature readings at any locations of interest on a die at run time.

Another issue to be resolved is the accuracy of thermal sensors; a thermal sensor in a 0.35μm 2.5 V digital CMOS technology, which was implemented in a general purpose microprocessor in the late 1990s, had the reading accuracy of ±12 ℃ with a resolution of 4 ℃, [96]. Since then, great improvement has been made in its reading accuracy, and recent sensors report accuracy of ±1 [63], but there still remains a lot of work to be done especially when it comes to the design of on-chip thermal sensors that are fully compatible with digital CMOS technologies.

27.3.4 Sensor Allocation: Hotspot Monitoring

Since the late 1990s and the early 2000s, thermal distribution of a chip has become a lot more complicated due to a large number of hotspots spreading across a die, and multiple thermal sensors came into play to monitor the temperatures of hotspots more efficiently and accurately.

One simple way to place multiple thermal sensors on a die is to place them on a uniform grid. As a result, some hotspots might not be detected, and the accuracy will be quite limited especially when a small number of thermal sensors is used. Linear interpolation technique using the temperature readings of four neighboring thermal sensors was proposed for the estimation of the maximum temperature of a chip [69];

When thermal distribution of a chip is available, this information can be used for sensor allocation, and thermal sensors can be allocated in a smart way so that the hotspots of a chip can be monitored correctly while minimizing the number of thermal sensors. In [72], a sensor allocation algorithm divides the die area into an array of blocks using the information on hotspot locations, and the size of each block is adjusted in such a way that all hotspots in each block can be covered and monitored by a single thermal sensor assigned to the block. This method works well when the number of hotspots is not large, but with the increase in the number of hotspots, a lot more thermal sensors will be required.

A thermal sensor allocation method based on k-means clustering [65] was proposed in [71]; each and every hotspot is assigned to one of k clusters recursively, where k is the number of thermal sensors, so that the Euclidean distance between the centroid of a selected cluster and the hotspot is minimized. Then, k thermal sensors are assigned to the centroids of those k clusters. However, this method might

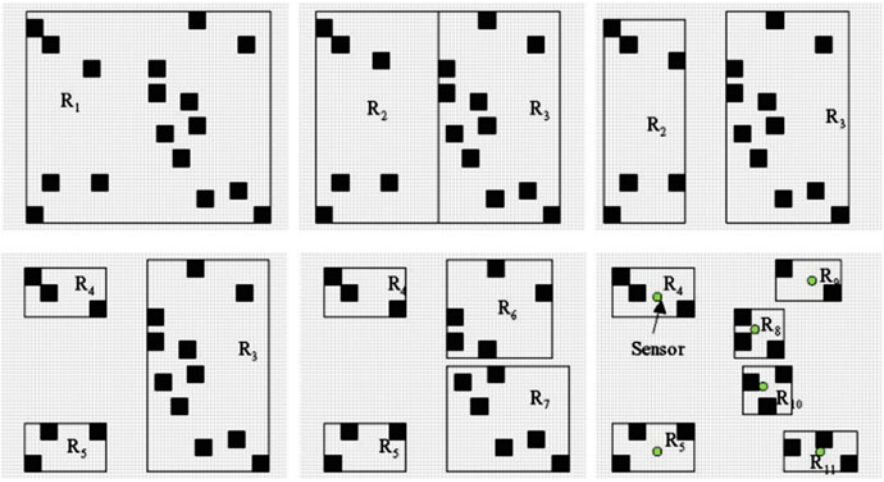


Fig. 27.15 Recursive bisection based thermal sensor allocation [72]

produce some unreasonable results especially when remotely located hotspots have smaller temperature differences than closely located hotspots. Figure 27.15 shows an example of such systems.

27.3.5 Sensor Allocation: Full-Chip Profile Reconstruction

In recent years, a large number of new sensor allocation methods have been proposed to support full-chip thermal profile reconstruction at run time from the temperature readings of a small number of sensors. Sensor allocation is performed with a view to a better run-time thermal profile reconstruction from the beginning, and the number and the locations of thermal sensors are determined accordingly. Fine-grain DTM solutions can make full use of the detailed temperature information from full-chip profile reconstruction, especially on multi-core processors [88]; task migration among cores can be performed more efficiently, and the thermal behavior and static power consumption of caches, which consume a large portion of the die area, can be optimized [47, 49].

In [77] (Fig. 27.16), energy analysis in frequency domain was used for sensor allocation; the main idea of this method is that thermal sensors should be distributed in proportion to the high-frequency energy in frequency domain so that more sensors can be assigned to regions with large thermal variations. This method alternates vertical bisection and horizontal bisection, and then compares the high-frequency energy of the two bisected regions. Thermal sensors are allocated proportionately, and the bisection continues until all thermal sensors are assigned.

In [112], a statistical methodology was developed for sensor allocation and full-chip thermal profile reconstruction; the entire die area was divided into a 16-by-16

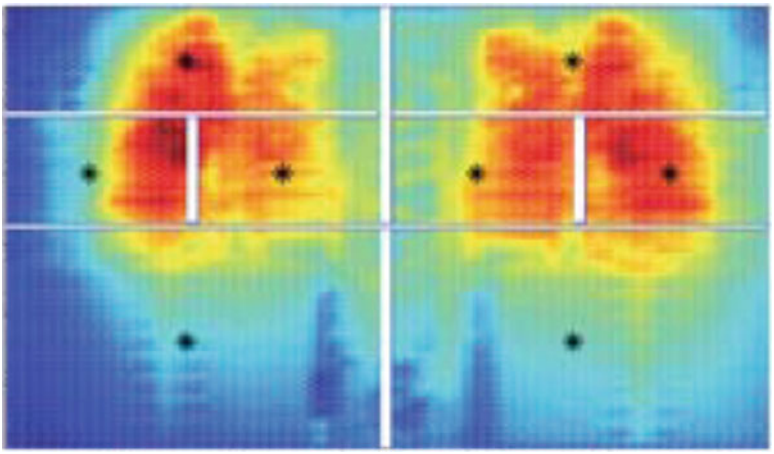


Fig. 27.16 Energy-aware thermal sensor allocation [77]

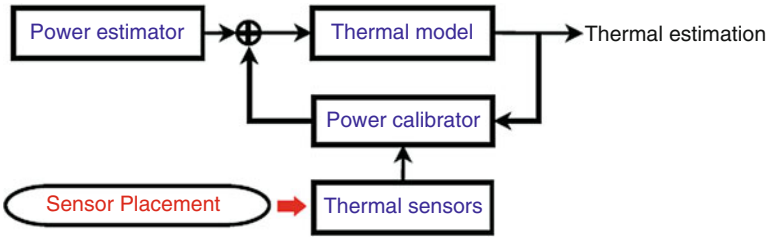


Fig. 27.17 Thermal profile estimation based on sensor-assisted power estimation [107]

grid, and a set of nodes on the grid were selected so that the thermal correlation among them can be minimized, and the thermal correlation between the selected nodes in the set and the nodes outside the set can be maximized at the same time. In this way, each thermal sensor can provide as much temperature information as possible on the non-sensor nodes, while the redundancy among the sensor nodes is minimized.

One way to have accurate temperature information of a chip is to solve the heat differential equation directly with correct power information [43]. Performance counter-based run-time power estimators [86, 108] can be used to supply power information at run time, but they tend to have some power estimation errors. A new approach to achieve good temperature estimation based on the differential equation was proposed in [107] (Fig. 27.17), and it exploits the temperature readings of thermal sensors to correct the power estimation errors. According to the simulation results on a dual-core processor and SPEC2000 benchmark suites [3], it achieved the maximum error of 1.2 ℃, and the averaged error of 0.085 ℃ with six thermal sensors.

In [101] a novel approach of using multiple virtual thermal sensors to increase the accuracy of temperature readings was presented; the virtual thermal sensors are generated from a small low-power physical thermal sensor by adaptively switching its calibration points on the run. Simulation results show that the RMS error of temperature readings can be reduced by up to 91.1% with the use of four virtual thermal sensors as compared with a single thermal sensor case.

27.4 Reliability Modeling

Modern highly scaled CMOS circuits suffer from performance and power losses due to short channel effects that exacerbate process variations. Process-induced variations are typically classified as either systematic or random variations. Systematic variations are predictable in nature and depend on deterministic factors such as layout and surrounding topological environment [74]. These types of errors are handled by static redundancy techniques. Random variations, on the other hand, pose one of the major challenges in circuit design in the nanometer regime [12]. This variation shifts the process parameters of different transistors in a die in different directions, which can result in significant mismatch between neighboring transistors [66]. This phenomenon is typically referred to as Random Dopant Fluctuations (RDF). RDF has the dominant impact on the transistors strength mismatch and is the most noticeable type of intra-die variation that can lead to cell instability and failure. These failures are manifested as either an increase in the cell access time or unstable read and write operations. Typically, RDF effects are countered by increasing the operational supply voltage, thus effectively masking away any variations in the individual transistor threshold voltage. Clearly, this leads to higher power consumption. One major aspect of RDF is that the randomness of the variations results in a random distribution of the access errors across the two-dimensional area of a memory [12, 66]. This phenomenon is a key element that can be exploited by cross layer approaches, since random errors are much easier to handle at higher layers of the system via error correction techniques such as data redundancy (coding) or spatial and temporal filtering. Several research efforts considered exploiting this phenomenon by reducing the supply voltage (i.e., Voltage Over Scaling (VOS)) [6, 44, 73, 75, 76, 99], while informing higher network layers of the anticipated increase in memory fault rates.

27.4.1 Memory

Figure 27.18 shows the typical six-transistor cell used for CMOS Static Random-Access Memories (6T SRAM). The cell consists of two cross-coupled CMOS inverters (NL,PL and NR,PR) that store one bit of information, and two N-type transistors (SL and SR) that connect the cell to the bitlines (BLC and BLT). Classically [66] failures in memory cells are categorized as either of a transient nature dependent on operating conditions or of a fixed nature due to manufacturing

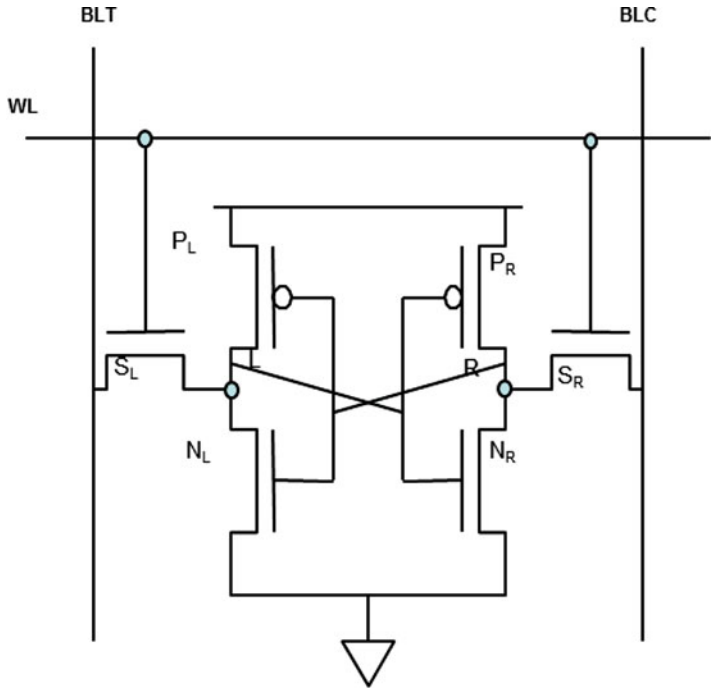


Fig. 27.18 6T SRAM Cell

errors. Symptoms of these failures are expressed as (1) increase in cell access time, (2) write time, or (3) unstable read operations. Conventionally, we assumed that fixed errors are predominant, with a minority of the errors introduced due to transient effects. In sub-100 nm designs, RDF has the dominant impact on the transistors strength mismatch and is the most noticeable type of intra-die variation that can lead to cell instability and failure in embedded memories. RDF has a detrimental effect on transistors that are colocated within one cell, by creating a mismatch in their intrinsic threshold voltage Vt. Furthermore, these effects are a strong function of the operating conditions (voltage, frequency, temperature, etc.).

The total cell failure probabilities for different $\mathbf{V}_{dd}$ as a function of T_{Max}, $P_T[Fail] = Prob[RAF \cup WF \cup DRF]$, are shown in Fig. 27.19 where RAF is read access failure, WF is write failure, and DRF is destructive read failure [31]. This figure illustrates that designers can trade off $\mathbf{V}_{dd}$, performance and error (failure) tolerance to achieve an optimal solution for a given set of conditions. It is important to make a distinction between errors and performance. Performance here is taken to mean achieving a specific ($\mathbf{T}_{Max}$) target as a predefined speed for the SRAM cell, while errors are taken to be hardware malfunction such as **RAF**, **WF**, etc. Intuitively, for a specific performance target, the designers can trade off error tolerance versus supply voltage. In other words, to achieve a low power solution, the designer must first decide on the acceptable level of error tolerance that is

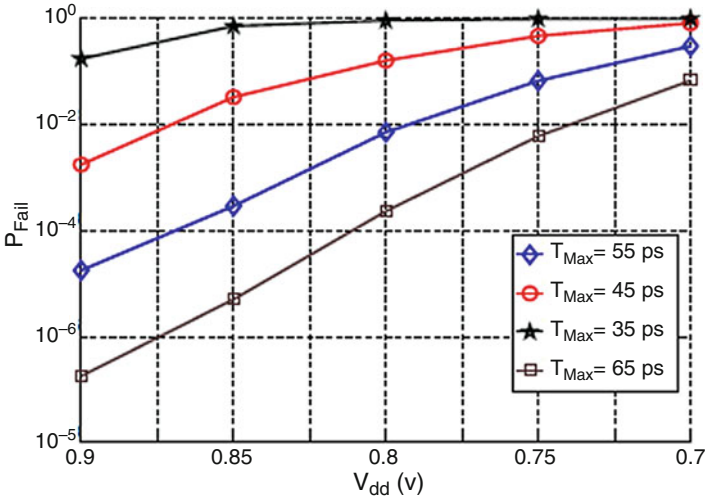


Fig. 27.19 Total cell failure probabilities

permissible by the application and the overall system design while still maintaining the required performance. Given that level, and a required performance level (i.e., $\mathbf{T}_{Max}$), the designer can select the appropriate $\mathbf{V}_{dd}$ from Fig. 27.19. For instance, consider the case that a wireless receiver using a Turbo Decoder is working at nominal $\mathbf{V}dd = 0.9\,v$ and at the failure rate of 10^{-7} and delay of $\mathbf{T}_{Max} = 65\,ps$. It has been shown in [50] that this system can handle memory errors up to 0.1% (10^{-3}). From Fig. 27.19, one can find out that by dropping the $\mathbf{V}_{dd}$ from 0.9 v to almost 0.775 v, the error is still less than 10^{-3} and the system can work at the same performance level, but at lower power.

27.4.2 Combinational Logic

Unlike memory, the propagation delays (t_{pd}) of arithmetic and logic circuits are highly dependent on the input patterns to the block and its circuit implementation [110]. Therefore, errors are not spatially random and one cannot find a closed form failure model for arithmetic and logic circuits. Applying VOS to logic and arithmetic blocks introduces input-dependent errors (timing violations) at the circuit level. Consider a logic gate Z with two inputs a and b and output x. To characterize this gate, the transistor-level circuit representing gate Z (or extracted from layout for more precise modeling of parasitics) is implemented in a Spice simulation making similar assumptions about threshold voltage, Vth as in Section 27.4.1. A Monte-Carlo simulation is run on the circuit for each of the possible 2^{2n} input-vectors, where n is the number of input signals to the gate, and an input-vector consists of the previous and current states of the inputs. The propagation delays

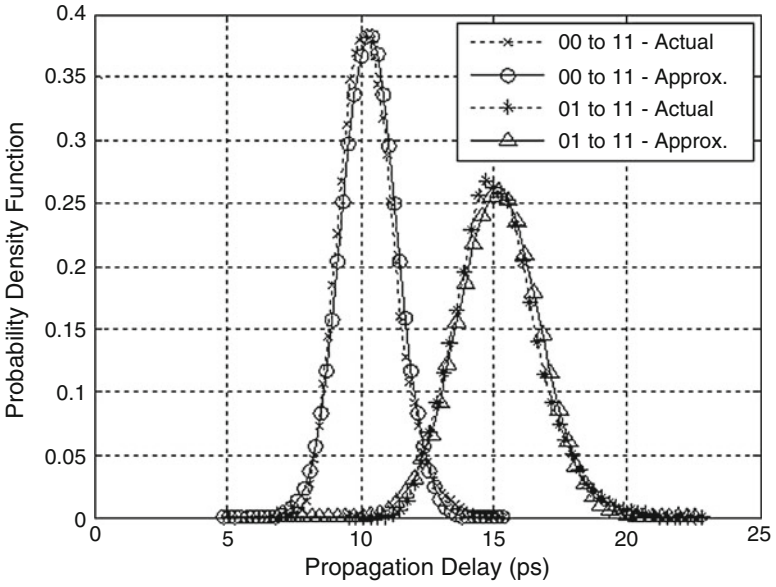


Fig. 27.20 pdf of a 2 input CMOS AND gate

statistics and the average power consumption for each input-vector are measured and stored. Figure 27.20 shows the Probability Density Functions (PDFs) of the propagation delays of a two input CMOS AND gate simulated in a 32 nm process under nominal supply voltage of 0.9 V using predictive transistor-models [1]. Two input-vectors, with input state transitions $ab = 00 \rightarrow 11$ and $ab = 01 \rightarrow 11$, are used. The PDFs of the measured propagation delays for the two scenarios show a very close match as compared a normal distribution approximation $\mathcal{N}\left(\mu_i, \sigma_i^2\right)$. Note that even though the outputs of the two input vectors are the same, their propagation delays are considerably different because of the initial state of the inputs. As circuit size increases, the complexity of modeling such delay distributions quickly becomes unmanageable. To address this challenge, one needs to incorporate circuit-level failures into a system-level simulation. While Statistical Static Timing Analysis (SSTA) [13] rapidly gives useful statistics of propagation delays and timing violations of critical paths, it does not give any information about the specific *input-vectors* that will cause timing violation errors in those paths. Therefore, SSTA cannot be used to address this challenge. On the other hand, Dynamic Timing Analysis (DTA) [61, 106] simulates circuits for functionality to acquire propagation delays on a per input-vector basis. Hence, DTA can be used to address the challenge of trading off reliability versus energy efficiency. In doing so, one can attempt to integrate a circuit simulator (such as Spice) into the system-level simulation to acquire propagation delay results on a per input-vector basis. This, however, will be very costly in terms of processing overhead and simulation time, since the quality and accuracy of DTA is directly proportional to the number of input

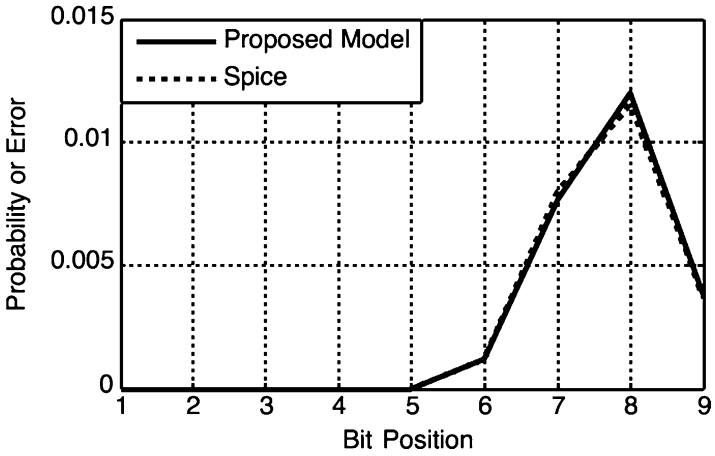


Fig. 27.21 Comparing probability of error per bit from proposed model and from Spice simulation

test vectors used. Simulating a simple digital block for one input-vector in Spice requires run time in the order of few hundred milliseconds. This would be very inefficient for processing large amounts of data. Methods such as [110] attempt to macromodel these distributions and propagate them in a consistent way, allowing the modeling of large combinational components such as Adders, multipliers, and CORDIC. Figure 27.20 compares the probability of error per bit for the adder from the proposed model and from the Spice simulation, and it confirms that the second most significant output bit has the highest probability of error (Fig. 27.21).

27.4.3 Microarchitecture and System Level

In recent years, numerous research efforts have targeted reliability-power-performance trade-offs via software, microarchitectural, and circuit techniques, including several efforts by the Principal Investigators (PIs) as outlined above. For example, in [91] Rinard et al. identify inherent redundancies in computational patterns such as sum and mean calculations and indicate the insignificant impact of resource reduction on such patterns. Moreover, techniques such as varying clock frequencies, skipping tasks, loop perforation, dynamic knobs, and using alternative implementations for key components [7, 9, 41, 70, 90] have also been performed with minimal effect on the accuracy of final performance metrics. Other approaches [24, 104] propose relaxing the correctness at the end results. These approaches allow the user to determine the minimum necessary precision needed at a given point or output of the code and adjust the amount of calculations performed to only satisfy the required accuracy. To use the hardware redundancies, the EnerJ approach [95] introduces an approximated language that enables the developer to

distinguish the precise and "approximatable" parts of the code and save energy on portions that can tolerate approximation. Furthermore, a more detailed work by the same research group [110] uses EnerJ as the guarantee for reliability and proposes language constructs that focus more on power-saving possibilities in a pipelined architecture and further specify the hardware implementations of such approximated language. ERSA [59] is a more drastic technique for saving power on a multi-core architecture. It divides the cores to reliable and unreliable cores and proposes to reduce the voltage on all parts of the unreliable cores. Even though there is no error recovery method introduced and the frozen cores are restarted by reliable ones, the output remains more than 90% accurate for the tested set of benchmarks. Earlier works such as the Aura/Odyssey/Coda project [76] investigated mobility and adaptation at the software level via what was termed "application-aware adaptation." Finally, the work of Breuer and Gupta promotes the concept of living with processing errors in some cases in order to improve yield [16, 17].

On the microarchitectural front, researchers have proposed several approaches that attempt to exploit architectural innovations to reduce excessive design margining associated with process variations. Examples include Razor [30, 35, 57] and TEAtime [105] that add extra hardware to correct for errors. The research in [76, 99], and [31] promoted the use of "algorithmic noise tolerance" and proposed using adaptive filters and replication to minimize the impact of scaling V_{dd} beyond the critical region for basic DSP functions, e.g., filtering and other communication blocks. The work proposed in [73] and [75] considers faulty caches and means of dealing with process faults through isolating faulty cache lines. On the circuits front, there has been significant work to achieve low-power operation through a combination of circuit design and technology-dependent optimizations [17, 21–23, 28, 51, 52, 68, 81, 97].

27.5 Interplay between Power, Temperature, Performance, and Reliability

The power, temperature, performance, and reliability of a chip exhibit a complex relationship where a small change in one dimension can potentially affect other characteristics. In order to illustrate the complexity of the interacting metrics or power, performance, and reliability in a dynamically changing environment, we present as an example an embedded memory block. Classically, failures in embedded memory cells are categorized as either of a transient nature (because of operating conditions) or of a fixed nature (due to manufacturing errors). Failures are manifested as (1) increase in cell access time, or (2) unstable read/write operations. In process technologies larger than 100 nm, fixed errors are predominant, with a minority of the errors introduced due to transient effects. This model cannot be sustained as scaling progresses due to the random nature of the fluctuation of dopant atom distributions. In fact, in sub 100 nm design, RDF has the dominant impact on the transistor's strength mismatch and is the most noticeable type of intra-die variation that can lead to cell instability and failure in embedded memories [55].

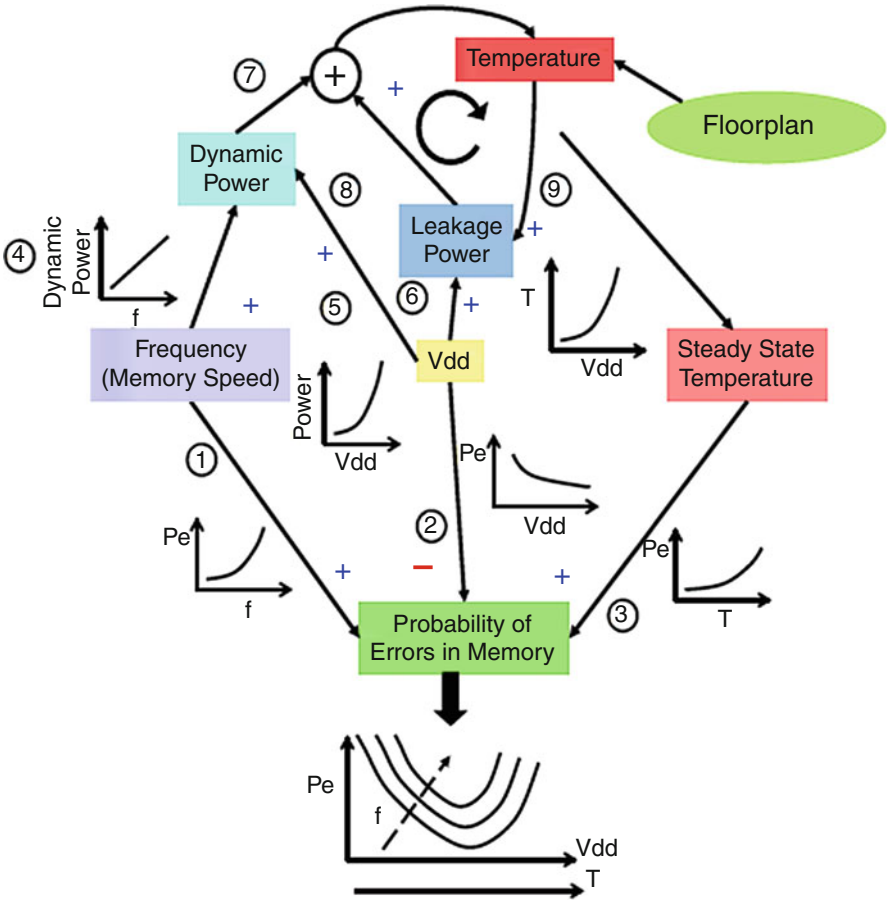


Fig. 27.22 Sensitivity of memory errors to various parameters

RDF has a detrimental effect on transistors that are colocated within one cell by creating a mismatch in their intrinsic threshold voltage, V_t. Furthermore, these effects are a strong function of the operating conditions such as voltage, frequency, temperature, etc.

Figure 27.22 shows how errors in memory are affected by different parameters. As the operating frequency is increased, the probability of memory errors increases (1) because it enforces tighter bounds on the time allowance for memory accesses. Increase in V_{dd} reduces the cell delay and thus causes the errors to decrease (2). The errors in memory increase along with the rise in temperature (3) because of increase in the cell delay. These are not the only relationships that affect memory errors. From Fig. 27.22, we also examine other interrelationships at work: The dynamic power dissipation in memory cells increases with increase in both frequency ($\propto \quad f$) and V_{dd} ($\propto V_{dd}^2$) . The leakage power, on the other hand,

increases with V_{dd} ($\propto e^{\beta V_{dd}}, \beta > 1$). Both dynamic power and leakage power determine the operating temperature. Leakage power dissipation of a cell is known to increase superlinearly with increase in temperature. As temperature increases, the leakage power dissipation increases which further elevates the temperature. This "positive feedback loop" between temperature and leakage power stabilizes when steady-state operating temperatures have been reached at which state, all the dynamic and leakage power dissipation is transferred to the environment by the package [30]. This discussion implies that probability of error is not a monotonically decreasing function of supply voltage but rather exhibits a convex behavior as shown in Fig. 27.23. A comprehensive approach to memory/logic design must consider these mutual interdependent relationships. The effect of interplay between V_{dd}, probability of error, and temperature for different cell speeds is shown in Fig. 27.23 where the different curves represent the behavior of memories with maximum allowed times of 70, 67, 65, and 60 ps, respectively. We observe that as the frequency of the cell increases (or the delay decreases), the probability of error also increases. We also observe that an increase in V_{dd} reduces the probability of error but only up to a certain point (marked **X**). After **X**, the rise in temperature due to V_{dd} increases the memory errors. However, for curve we do not observe this behavior because the speed of the cell is low and the probability of failure is not detectable by our simulation setup. In the absence of thermal considerations, these curves would have continued to exhibit decreasing probabilities of error with increasing V_{dd}.

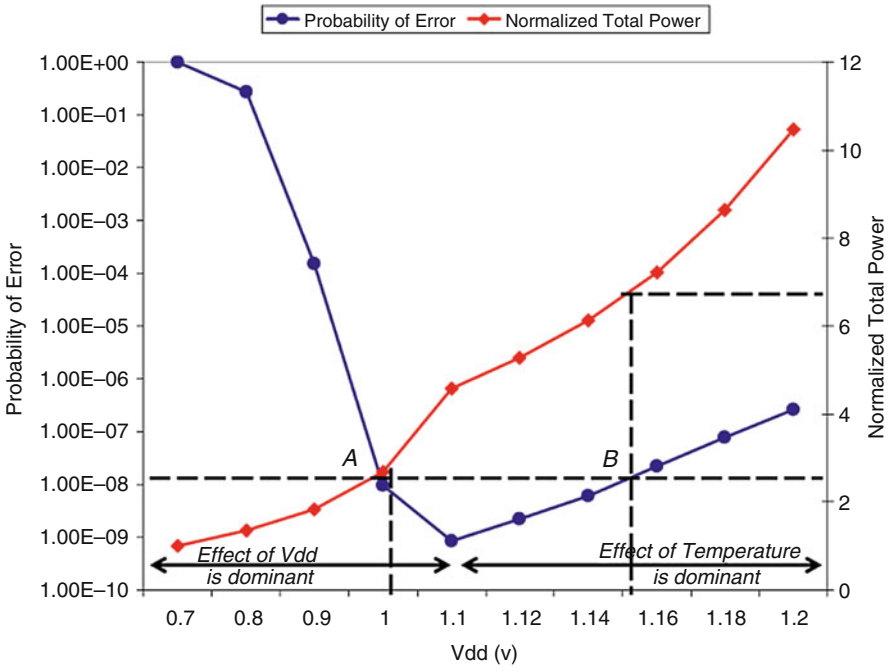


Fig. 27.23 Probability of error for different frequencies

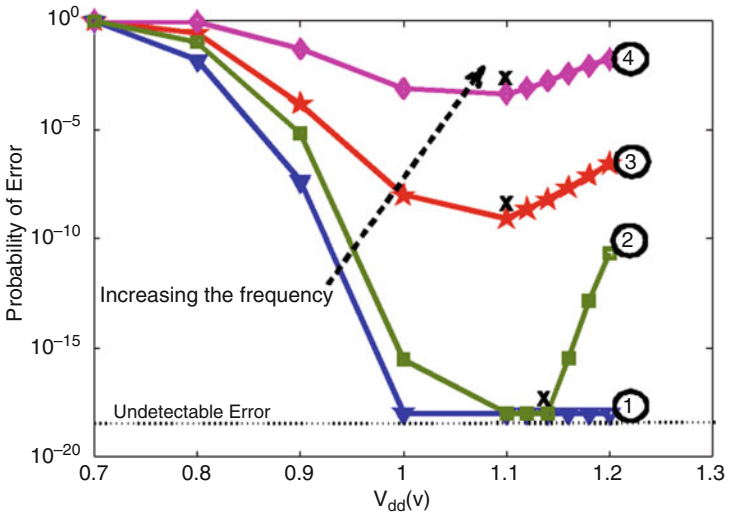


Fig. 27.24 Probability of error and total power

Figure 27.24 shows the normalized total power dissipation and the probability of error for a cell with maximum allowed time of 65ps. Initially, the effect of increase in V_{dd} is dominant and probability of error decreases with increase in V_{dd}. However, at higher V_{dd} the effect of resulting temperature becomes dominant and probability of error increases with increase in V_{dd}. The figure illustrates that for a given probability of failure target, two voltage levels can be chosen that achieve the desired target. For example, at a target probability of error of 10^{-8}, one can select either 1.0 v (Point A) or 1.16 v (Point B). However, the dynamic power at 1.0 v is 34.5% less than that at 1.16 v because dynamic power $\propto V_{dd}^2$. Even without thermal dependence, the leakage power at 1.0 v is 46.1% less than that at 1.16 v because leakage power $\propto e^{\beta V_{dd}}, \beta > 1$. Thus, the total power at 1.0 v is 2.5× less than that at 1.16 v. Designers can save significant power by operating at lower V_{dd} voltages while maintaining performance levels.

27.6 Power, Performance, and Resiliency Considerations in SoC Design

While scaling V_{dd} is indeed one of the most effective means of controlling power, it is imperative to understand how other effects can be factored in. For example, incorporating temperature, process variations, etc. will lead to significantly different system policies than those which would be adopted in a non-cross layer aware approach. Figure 27.25 highlights the different phases in error development as V_{dd} is scaled. When V_{dd} is close to its typical value, the system operates normally. As we scale down, errors start to develop. Depending on the system, such errors may

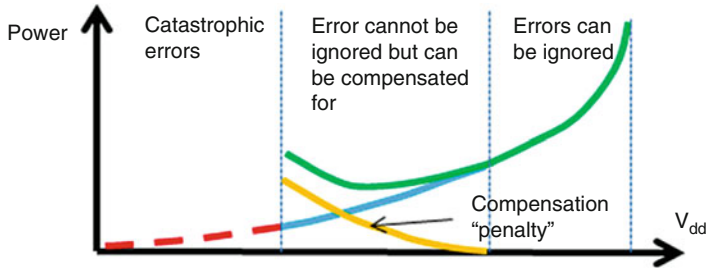


Fig. 27.25 Different phases in error development

be ignored, but only up to a point beyond which it becomes necessary to deploy measures that will compensate, either partially or fully, for the drop in quality. Discussed below, these measures can occur at different layers of abstraction and incur a penalty causing a lessening of the power savings due to V_{dd} scaling. This may lead to a "sweet spot" at which maximum power savings can be achieved. If we keep decreasing V_{dd}, not only would the savings become less, but there typically exists a point at which errors can become so large that the system breaks down and quality cannot be recovered.

How much errors can be tolerated depends on the technology, architecture, and application. Similarly the method of error compensation depends on these parameters as well and can be applied at all the design layers. In the following, we present examples of error tolerance at these various design layers and methods of compensation. These examples serve as data points and are not intended to limit the scope of the discussion.

27.6.1 Architecture-Level Error Tolerance

Consider the case of a stringently error-constrained system. A representative 16 KB processor cache is assumed, with a block size of 16 bytes and associativity of 1, based on an underlying 70 nm CMOS technology. Cache errors can either lead to unstable system behavior or excessive delay due to a cache miss. By varying the voltage of the cache from 0.9 to 0.7 v, it was observed that the miss rate increases from 3.9 to 59.5%, respectively. It is clear that reducing the supply voltage increases parametric errors to an extent where the memory becomes useless. To counter this effect, a modified structure allows the memory to continue operation with minimal impact on the miss rates even at elevated error rates [50]. The proposed architecture is shown in Fig. 27.26 (Red dots indicate parametric errors), where, in addition to the cache block, two other blocks are added, the Bit Lock Block (BLB) and the Inquisitive Defect Cache (IDC). The BLB is an off cache defect map that stores a tag bit to identify faulty locations. These can be updated via a built-in self-test initiated at each configuration update. The IDC acts as a place holder for defective cache words. Due to the random spatial distribution of RDF induced faults, as well

Benchmark	32nm		65nm	
	Min Voltage	Power Saving	Min Voltage	Power Saving
DEC	0.75	43.25%	0.84	10.23%
LISP	0.72	49.35%	0.77	18.36%
MACR	0.75	44.32%	0.83	12.21%
PASC	0.74	45.60%	0.83	11.90%
MUL	0.75	43.12%	0.84	7.93%
SPIC	0.72	44.73%	0.77	15.21%

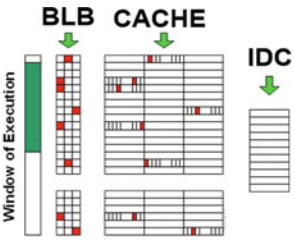


Fig. 27.26 IDC Error-tolerant cache architecture

as the locality of access in a cache, the size of IDC cache could be much smaller than the total size of all the defective words in a cache. Since this cache is masking parametric errors, it can be thought of as a means of tightening the distribution of faults as a function of the change in supply voltage. In this sense, DMS can be used to identify the optimum size of this cache based on an expected distribution of errors due to supply changes. If the IDC size and associativity are chosen properly, the execution window of a process in the cache (after its first pass) will experience very little defective/disabled words.

Both the IDC and the BLB can be assumed to be operated at a higher and safer voltage (within temperature constraints) or utilizing larger devices, while Adaptive Supply Voltage (ASV)/Adaptive Body Bias (ABB) is applied on the cache body. Early simulation results, using standard benchmarks, and a trace-driven simulator are shown in Fig. 27.26 where the miss rate is reduced down to 6.45% (from 59.5% initially) and power savings of more than 40% are reported. The interested reader is referred to [51] for more details about the simulation setup and results. This approach benefits from the spatial randomness of process-induced faults, thus allowing the use of a very small victim cache to perform cache remapping at elevated error rates without reducing the size of the cache.

27.6.2 Application-Level Error Resiliency: Multimedia Applications (H.264)

Consider an H.264 system Fig. 27.27 (left) as a representative application for mobile multimedia systems. One of the biggest challenges is power consumption, which is typically addressed by power management, mainly by reducing the supply voltage. However the range of such a reduction is limited by (1) performance constraints and (2) component reliability under very low V_{dd}.

By design, these systems have built-in error resiliency that has been exploited in many different compression and transmission schemes mainly as a quality trade-off. [54] proposed utilizing aggressive voltage scaling on *embedded memories* resulting in low-power, high-frequency operation, albeit, with errors due to scaling. Based on the error statistics, we propose, analyze, and quantify the performance and overhead (in terms of power and area) of various filtering and mapping techniques

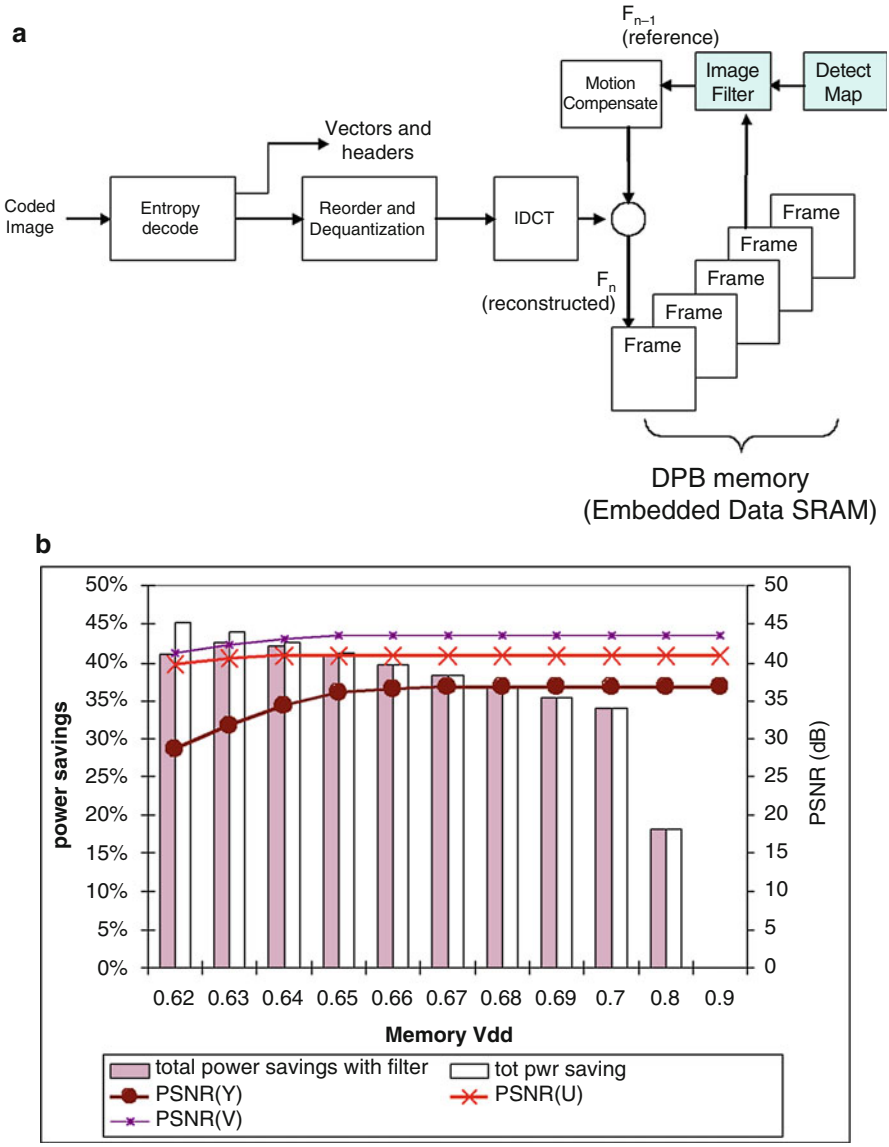


Fig. 27.27 H.264 decoder with filtering (*left*) & Image quality (PSNR) vs power savings at different Vdd levels (Foreman Video) (*right*)

that compensate for the errors, thus enabling the system to operate at lower voltages while meeting system specifications. Finally, we quantify the expected system power savings due to the above mentioned approach. Figure 27.27 (right) shows the results of such an exploration. When we lower V_{dd} on the decoder memories, its reliability decreases and as a result, the output quality drops. This

can be compensated for by filtering, which consumes power so the gains from V_{dd} reduction tend to lessen as error rates increase. However, the overall results indicate that good performance (PSNR) can be maintained even at very low V_{dd} while saving over 40% in overall system power consumption. While this case study highlights a significant opportunity in power savings, it requires an important paradigm shift in today's system design flow. Current flow emphasizes compartmentalization between system-level designers and backend (chip) designers, thus necessitating 100% correctness in hardware. The new paradigm de-compartmentalizes this flow and allows system designers to be aware of the physical layer through model abstractions.

27.6.3 Application-Level Error Resiliency: Wireless Modem Application (WCDMA)

In this section, we now extend the discussion to the transmission medium, using Wideband CDMA (WCDMA) as a representative of a wireless physical layer. Figure 27.28 depicts the top-level block diagram of a diversity enabled WCDMA SoC modem [34]. The SoC includes the modem section (RAKE receiver), the coding layer and the protocol layer of the standard. It is based on a dual embedded microcontroller architecture. The symbols from the modem are soft values with 10-bit precision that are available for all the data and control symbols transmitted on the data channels. Naturally, control symbols are very important and thus must be stored in a protected memory with minimum loss. However, data symbols possess a high degree of redundancy typically inserted by the channel coding scheme. Specifically in WCDMA [50], both Turbo and Viterbi schemes are supported. Thus, the data

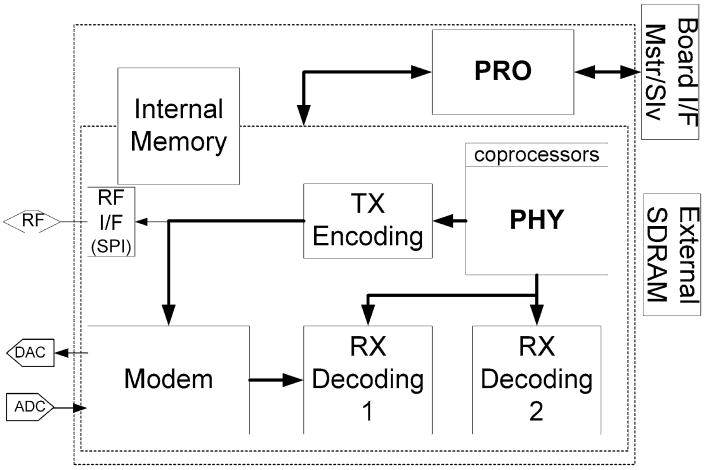


Fig. 27.28 WCDMA chip architecture

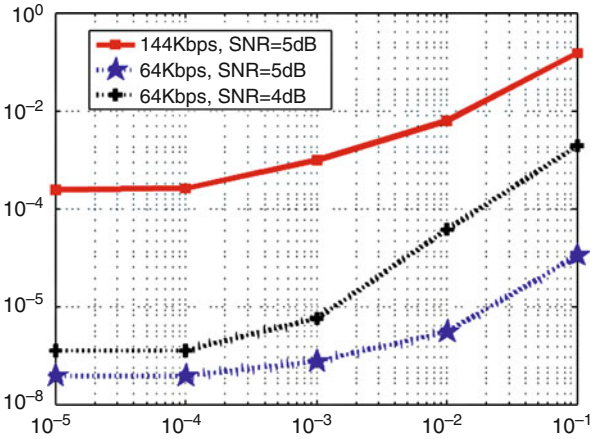


Fig. 27.29 Effect of the WCDMA memory errors on the system Bit Error Rate

memory can be partitioned into defect tolerant and non-defect tolerant sections. A defect tolerant memory is a memory that is used primarily to buffer data and thus can be a target of aggressive power management. It is interesting to note that the data buffering memories (defect tolerant candidates) consume approximately 50% of the overall memory required for the entire modem.

Figure 27.29 shows the effect of the memory errors on the WCDMA Bit Error Rate (BER) for different transmission bit rates. As expected, given the same SNR, for higher bit rates, the memory errors have higher impact on the BER since there is less redundancy in the system. A power analysis of the architecture indicates that the overall memory consumes roughly 45% of the total power. In prior work, the PIs have shown that by applying error aware dynamic voltage scaling a savings of 46% in leakage power and 44% in dynamic power is possible in the error-tolerant memories. It is important to note that these savings are independent of other power-saving methods such as reducing frequency of operation, etc.

27.6.4 Mobile Phone SoC Example

The previous two case studies illustrated the design space exploration for a wireless and a video application. In today's mobile phones, these applications are synergistic and are typically implemented on the same chip. Figure 27.30 illustrates a hypothetical modern mobile phone SoC with a WCDMA physical layer to convey the data stream and an H.264 video codec as the application. While in the previous cases we considered each application separately, in this case, the question arises as to which of the two applications to target for V_{dd} reduction in order to save overall SoC power? To do so we investigated three scenarios: Case (A) Nominal-V_{dd} for the WCDMA modem and aggressive dynamic voltage scaling (AVS) for the H.264

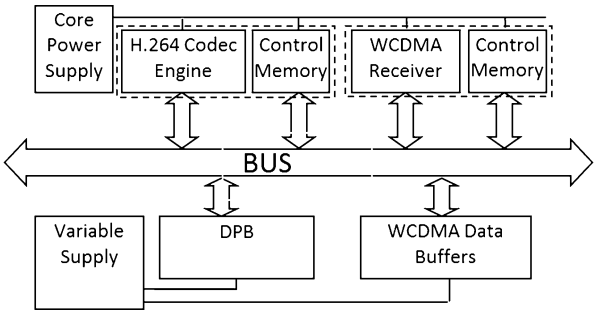


Fig. 27.30 A mobile phone SoC architecture

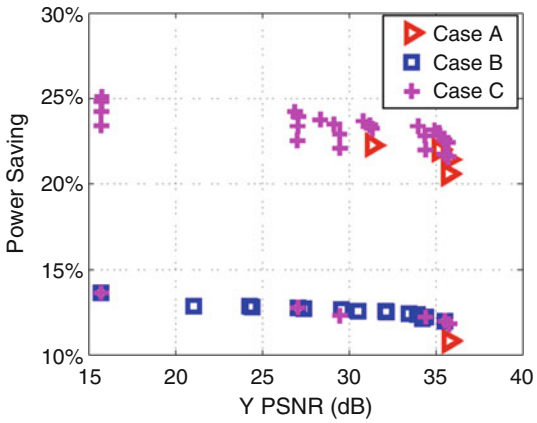


Fig. 27.31 Application-aware design space exploration for the cell phone SoC

decoder engine, Case (B) supply scaling for the WCDMA modem and nominal-V_{dd} for the H.264 decoder, and Case (C) supply scaling for both the H.264 decoder and the WCDMA modem. Based on prior experience, the PIs have with WCDMA systems, it is estimated that the WCDMA modem consumes 72% of the total power whereas the H.264 decoder consumes 28% of the total power in 65 nm technology node [50]. As this portion changes, the gains will scale accordingly. Figure 27.31 illustrates a summary of the results depicting the expected power savings for each case versus the luma (Y) component of the image as a quality metric. From the graph, we observe that some points are inferior to others. In other words, one case may yield higher power savings than another for the same target PSNR. Such points are considered as Pareto-optimal. Overall, Case B appears to be inferior to cases A and C. However, this is a direct result of the ratio of power consumption of the receiver and video decoder. Since the receiver consumes more than 3× the power of the H.264 decoder, one would expect to get more power reduction by supply scaling of the receiver. This situation would be reversed in another system where the ratios are the opposite. As expected, case C yields the most Pareto-optimal design points since it is a superset of cases A and B.

27.7 Summary and Conclusion

This chapter presented a typical design flow for integrating microarchitectural IPBs into complex SoCs that must satisfy performance, power, thermal, and reliability constraints. Toward this end, we first presented different abstraction levels for SoC design that promote IP reuse and which enable fast simulation for early functional validation of the SoC platform. Since SoCs must satisfy a multitude of interrelated constraints, we then presented high-level power, thermal, and reliability models for estimating these constraints. We outlined the complex interrelationship between power, temperature, performance, and reliability of an SoC and illustrated these dependencies. We concluded the chapter with several case studies presenting examples of error tolerance at these various design layers and methods of compensation.

References

1. Predictive technology model(ptm). http://www.eas.asu.edu
2. Synopsys design compiler, primetime px, power compiler. http://www.synopsys.com
3. Standard performance evaluation council, performance evaluation in the new millennium, v.1.1 (2000)
4. Functional Specification for SystemC 2.0. www.systemc.org (2001)
5. International technology roadmap for semiconductors (2011) System drivers. Technical report.
6. Abdallah R, Shanbhag N (2009) Error-resilient low-power Viterbi decoder architectures. IEEE Trans Signal Process 57(12):4906–4917. doi:10.1109/TSP.2009.2026078
7. Ansel J, Chan C, Wong YL, Olszewski M, Zhao Q, Edelman A, Amarasinghe S (2009) Petabricks: a language and compiler for algorithmic choice. In: Proceedings of the 30th ACM SIGPLAN conference on programming language design and implementation, PLDI '09. ACM, New York, pp 38–49. doi:10.1145/1542476.1542481
8. ARM (2001) ARM AMBA specification and multi layer AHB specification, (rev2.0). http://www.arm.com
9. Baek W, Chilimbi TM (2010) Green: a framework for supporting energy-conscious programming using controlled approximation. In: Proceedings of the 31st ACM SIGPLAN conference on programming language design and implementation, PLDI '10. ACM, New York, pp 198–209. doi:10.1145/1806596.1806620
10. Baniasadi A, Moshovos A (2001) Instruction flow-based front end throttling for power-aware high performance processors. In: Proceedings of the 2001 international symposium on Low power electronics and design (ISLPED'01). ACM, New York, pp 16–21. http://dx.doi.org/10.1145/383082.383088
11. Bansal N, Lahiri K, Raghunathan A, Chakradhar S (2005) Power monitors: a framework for system-level power estimation using heterogeneous power models. In: 18th international conference on VLSI design, pp 579–585 doi:10.1109/ICVD.2005.138
12. Bhavnagarwala A, Tang X, Meindl J (2001) The impact of intrinsic device fluctuations on CMOS SRAM cell stability. IEEE J Solid State Circuits 36(4):658–665. doi:10.1109/4.913744
13. Blaauw D, Chopra K, Srivastava A, Scheffer L (2008) Statistical timing analysis: from basic principles to state of the art. IEEE Trans Comput Aided Des Integr Circuits Syst 27(4):589–607. doi:10.1109/TCAD.2007.907047
14. Black J (1969) Electromigration – a brief survey and some recent results. IEEE Trans Electron Devices 16(4):338–347

15. Blair J, Ghate P, Haywood C (1971) Concerning electromigration in thin films. Proc IEEE lett 59:1023–1024
16. Breuer M (2010) Hardware that produces bounded rather than exact results. In: 2010 47th ACM/IEEE design automation conference(DAC), Anaheim, pp 871–876
17. Breuer M, Gupta S, Mak T (2004) Defect and error tolerance in the presence of massive numbers of defects. IEEE Des Test Comput 21(3):216–227. doi:10.1109/MDT.2004.8
18. Brooks D, Martonosi M (2001) Dynamic thermal management for high-performance microprocessors. In: Proceedings of the 7th international symposium on high-performance computer architecture (HPCA'01). IEEE Computer Society, Washington, DC, p 171
19. Brooks D, Tiwari V, Martonosi M (2000) Wattch: a framework for architectural-level power analysis and optimizations. In: Proceedings of the 27th international symposium on computer architecture, Vancouver, pp 83–94
20. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: First IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, pp 19–24. doi:10.1109/CODESS.2003.1275250
21. Calhoun B, Chandrakasan A (2004) Standby power reduction using dynamic voltage scaling and canary flip-flop structures. IEEE J Solid State Circuits 39(9):1504–1511. doi:10.1109/JSSC.2004.831432
22. Calhoun B, Daly D, Verma N, Finchelstein D, Wentzloff D, Wang A, Cho S, Chandrakasan A (2005) Design considerations for ultra-low energy wireless microsensor nodes. IEEE Trans Comput 54(6):727–740. doi:10.1109/TC.2005.98
23. Calhoun B, Wang A, Chandrakasan A (2005) Modeling and sizing for minimum energy operation in subthreshold circuits. IEEE J Solid State Circuits 40(9):1778–1786. doi:10.1109/JSSC.2005.852162
24. Carbin M, Kim D, Misailovic S, Rinard MC (2012) Proving acceptability properties of relaxed nondeterministic approximate programs. In: Proceedings of the 33rd ACM SIGPLAN conference on programming language design and implementation, PLDI '12. ACM, New York, pp 169–180. doi:10.1145/2254064.2254086
25. Center for Embedded Computer Systems: SpecC system. http://www.cecs.uci.edu/~specc/
26. Chabloz J, Hemani A (2010) Distributed DVFS using rationally-related frequencies and discrete voltage levels. In: Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design (ISLPED'10). ACM, New York, pp 247–252. http://dx.doi.org/10.1145/1840845.1840897
27. Chakrabarti C, Gaitonde D (1999) Instruction level power model of microcontrollers. In: Proceedings of the 1999 IEEE international symposium on circuits and systems, ISCAS '99, vol 1, pp 76–79. doi:10.1109/ISCAS.1999.777809
28. Cho S, Chadrakasan A (2004) A 6.5-ghz energy-efficient BFSK modulator for wireless sensor applications. IEEE J Solid State Circuits 39(5):731–739. doi:10.1109/JSSC.2004.826314
29. Coskun AK, Rosing TS, Whisnant K (2007) Temperature aware task scheduling in MPSoCS. In: Proceedings of the conference on design, automation and test in Europe (DATE'07). EDA Consortium, San Jose, pp 1659–1664
30. Das S, Roberts D, Lee S, Pant S, Blaauw D, Austin T, Flautner K, Mudge T (2006) A self-tuning dvs processor using delay-error detection and correction. IEEE J Solid State Circuits 41(4):792–804. doi:10.1109/JSSC.2006.870912
31. Djahromi A, Eltawil A, Kurdahi F, Kanj R (2007) Cross layer error exploitation for aggressive voltage scaling. In: 8th international symposium on quality electronic design, ISQED '07, pp 192–197. doi:10.1109/ISQED.2007.53
32. Donald J, Martonosi M (2005) Leveraging simultaneous multithreading for adaptive thermal control. In: Proceedings of the second workshop on temperature-aware computer systems
33. Donald J, Martonosi M (2006) Techniques for multicore thermal management: classification and new exploration. ACM SIGARCH computer architecture news. 34(2). IEEE computer society
34. Eltawil A, Grayver E, Zou H, Frigon J, Poberezhskiy G, Daneshrad B (2003) Dual antenna UMTS mobile station transceiver asic for 2 mb/s data rate. In: IEEE international,

solid-state circuits conference on Digest of technical papers, ISSCC 2003, vol 1, pp 146–484. doi:10.1109/ISSCC.2003.1234242
35. Ernst D, Das S, Lee S, Blaauw D, Austin T, Mudge T, Kim NS, Flautner K (2004) Razor: circuit-level correction of timing errors for low-power operation. IEEE Micro 24(6):10–20. doi:10.1109/MM.2004.85
36. Gasteier M, Glesner M (1996) Bus-based communication synthesis on system-level. In: Proceedings of 9th international symposium on system synthesis, pp 65–70. doi:10.1109/ISSS.1996.565880
37. Gerards M, Hurink JL, Kuper J (2015) On the interplay between global DVFS and scheduling tasks with precedence constraints. IEEE Trans Comput 64(6):1742–1754
38. Gronowski P, Bowhill W, Preston R, Gowan M, Allmon R (1998) High-performance microprocessor design. IEEE J Solid State Circuits 33:676–686
39. Gunther S, Binns F, Carmean D, Hall J (2001) Managing the impact of increasing microprocessor power consumption. Intel Technol J 5:1–9
40. Herbert S, Marculescu D (2007) Analysis of dynamic voltage/frequency scaling in chip multiprocessors. In: ACM/IEEE international symposium on low power electronics and design (ISLPED). IEEE
41. Hoffmann H, Sidiroglou S, Carbin M, Misailovic S, Agarwal A, Rinard M (2011) Dynamic knobs for responsive power-aware computing. In: Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems, ASPLOS XVI. ACM, New York, pp 199–212. doi:10.1145/1950365.1950390
42. HP Labs (2015) CACTI – An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. http://www.hpl.hp.com/research/cacti/
43. Huang W et al (2006) Hotspot: a compact thermal modeling methodology for early-stage VLSI design. IEEE Trans Very Large Scale Integr (VLSI) Syst 14(5):501–513. doi:10.1109/TVLSI.2006.876103
44. Hussien A, Khairy M, Khajeh A, Amiri K, Eltawil A, Kurdahi F (2010) A combined channel and hardware noise resilient Viterbi decoder. In: 2010 conference record of the forty fourth Asilomar conference on signals, systems and computers (ASILOMAR), pp 395–399. doi:10.1109/ACSSC.2010.5757543
45. Im S, Banerjee K (2000) Full chip thermal analysis of planar (2-D) and vertically integrated (3-D) high performance ICs. In: International electron devices meeting 2000. Technical digest. IEDM (Cat. No.00CH37138), San Francisco, pp 727–730.
46. ITRS International roadmap of semiconductors. http://www.itrs.net/
47. John JK, Hu JS, Ziavras SG (2005) Optimizing the thermal behavior of subarrayed data caches. In: Proceedings of the 2005 international conference on computer design (ICCD'05). IEEE computer society, Washington, pp 625–630. https://doi.org/10.1109/ICCD.2005.81
48. Kavvadias N, Neofotistos P, Nikolaidis S, Kosmatopoulos K, Laopoulos T (2003) Measurements analysis of the software-related power consumption in microprocessors. In: Proceedings of the 20th IEEE instrumentation and measurement technology conference, IMTC '03, vol 2, pp 981–986. doi:10.1109/IMTC.2003.1207899
49. Kaxiras S, Ju Z, Martonosi M (2001) Cache decay: exploiting generational behavior to reduce cache leakage power. In: Proceedings of the 28th annual international symposium on computer architecture (ISCA'01). ACM, New York, pp 240–251. http://dx.doi.org/10.1145/379240.379268
50. Khajeh A, Cheng SY, Eltawil A, Kurdahi F (2007) Power management for cognitive radio platforms. In: Global telecommunications conference, GLOBECOM '07. IEEE, pp 4066–4070. doi:10.1109/GLOCOM.2007.773
51. Khajeh A, Eltawil A, Kurdahi F (2011) Embedded memories fault-tolerant pre- and post-silicon optimization. IEEE Trans Very Large Scale Integr (VLSI) Syst 19(10):1916–1921. doi:10.1109/TVLSI.2010.2056397
52. Khajeh A, Kim M, Dutt N, Eltawil AM, Kurdahi FJ (2012) Error-aware algorithm/architecture coexploration for video over wireless applications. ACM Trans Embed Comput Syst 11S(1):15:1–15:23. doi:10.1145/2180887.2180892

53. Kumar A, Shang L, Peh L, Jha NK (2008) System-level dynamic thermal management for high-performance microprocessors. IEEE Trans Comput-Aided Des Integr Circuits Syst 27(1):96–108
54. Kurdahi F, Eltawil A, Yi K, Cheng S, Khajeh A (2010) Low-power multimedia system design by aggressive voltage scaling. IEEE Trans Very Large Scale Integr (VLSI) Syst 18(5): 852–856. doi:10.1109/TVLSI.2009.2016665
55. Lahiri K, Raghunathan A, Lakshminarayana G, Dey S (2004) Design of high-performance system-on-chips using communication architecture tuners. IEEE Trans Comput Aided Des Integr Circuits Syst 23(5):620–636. doi:10.1109/TCAD.2004.826585
56. Lee KJ, Skadron K (2005) Using performance counters for runtime temperature sensing in high-performance processors. In: 19th IEEE international parallel and distributed processing symposium, pp 8. doi:10.1109/IPDPS.2005.448
57. Lee S, Das S, Pham T, Austin T, Blaauw D, Mudge T (2004) Reducing pipeline energy demands with local DVS and dynamic retiming. In: Proceedings of the 2004 international symposium on low power electronics and design, ISLPED '04, Newport Beach, pp 319–324.
58. Lee I, Kim H, Yang P, Yoo S, Chung EY, Choi KM, Kong JT, Eo SK (2006) Powervip: SoC power estimation framework at transaction level. In: Asia and South Pacific conference on design automation, pp 8. doi:10.1109/ASPDAC.2006.1594743
59. Leem L, Cho H, Bau J, Jacobson Q, Mitra S (2010) Ersa: error resilient system architecture for probabilistic applications. In: Design, automation test in Europe conference exhibition (DATE), pp 1560–1565. doi:10.1109/DATE.2010.5457059
60. Lienig J (2013) Electromigration and its impact on physical design in future technologies. In: Proceedings of the 2013 ACM international symposium on physical design. ACM, 2013
61. Liou JJ, Krstic A, Jiang YM, Cheng KT (2000) Path selection and pattern generation for dynamic timing analysis considering power supply noise effects. In: IEEE/ACM international conference on computer aided design, ICCAD-2000, pp 493–496. doi:10.1109/ICCAD.2000.896521
62. Lloyd JR (1991) Electromigration failure. J Appl Phys 69:7601–7604
63. Long J, Memik S, Memik G, Mukherjee R (2008) Thermal monitoring mechanisms for chip multiprocessors. ACM Trans Archit Code Optim (TACO) 5(2):9
64. Macii E, Pedram M, Somenzi F (1998) High-level power modeling, estimation, and optimization. IEEE Trans Comput Aided Des Integr Circuits Syst 17(11):1061–1079. doi:10.1109/43.736181
65. MacQueen J (1967) Some methods for classification and analysis of multivariate observations. In: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, vol 1, no 14
66. Makhzan M, Khajeh A, Eltawil A, Kurdahi F (2007) Limits on voltage scaling for caches utilizing fault tolerant techniques. In: 25th international conference on computer design, ICCD 2007, pp 488–495. doi:10.1109/ICCD.2007.4601943
67. Mamidipaka M, Khouri K, Dutt N, Abadir M (2004) Analytical models for leakage power estimation of memory array structures. In: International conference on hardware/software codesign and system synthesis, CODES + ISSS 2004, pp 146–151. doi:10.1109/CODESS.2004.240909
68. Markovic D, Stojanovic V, Nikolic B, Horowitz M, Brodersen R (2004) Methods for true energy-performance optimization. IEEE J Solid-State Circuits 39(8):1282–1293. doi:10.1109/JSSC.2004.831796
69. Memik SO, Mukherjee R, Ni M, Long J (2008) Optimizing thermal sensor allocation for microprocessors. IEEE Trans Comput Aided Des Integr Circuits Syst 27: 516–527
70. Misailovic S, Roy D, Rinard M (2011) Probabilistically accurate program transformations. In: Yahav E (ed) Static Analysis. Lecture notes in computer science, vol 6887. Springer, Berlin/Heidelberg, pp 316–333. doi:10.1007/978-3-642-23702-7_24
71. Mukherjee R, Memik SO (2006) Systematic temperature sensor allocation and placement for microprocessors. In: Proceedings of the 43rd annual design automation conference. ACM

72. Mukherjee R, Mondal S, Memik S (2006) Thermal sensor allocation and placement for reconfigurable systems. In: IEEE/ACM international conference on computer-aided design (ICCAD'06). IEEE
73. Mukhopadhyay S, Mahmoodi H, Roy K (2004) Statistical design and optimization of SRAM cell for yield enhancement. In: IEEE/ACM international conference on computer aided design, ICCAD-2004, pp 10–13. doi:10.1109/ICCAD.2004.1382534
74. Mukhopadhyay S, Kim K, Mahmoodi H, Roy K (2007) Design of a process variation tolerant self-repairing SRAM for yield enhancement in nanoscaled CMOS. IEEE J Solid-State Circuits 42(6):1370–1382. doi:10.1109/JSSC.2007.897161
75. Mukhopadhyay S, Mahmoodi H, Roy K (2008) Reduction of parametric failures in sub-100-nm SRAM array using body bias. IEEE Trans Comput Aided Des Integr Circuits Syst 27(1): 174–183. doi:10.1109/TCAD.2007.906995
76. Noble B (2000) System support for mobile, adaptive applications. IEEE Pers Commun 7(1):44–49. doi:10.1109/98.824577
77. Nowroz A, Cochran R, Reda S (2010) Thermal monitoring of real processors: techniques for sensor allocation and full characterization. In: Proceedings of the 47th design automation conference. ACM
78. Onouchi M, Yamada T, Morikawa K, Mochizuki I, Sekine H (2006) A system-level power-estimation methodology based on ip-level modeling, power-level adjustment, and power accumulation. In: Asia and South Pacific conference on design automation, pp 4. doi:10.1109/ASPDAC.2006.1594742
79. Orio Rd, Ceric H, Selberherr S (2010) Physically based models of electromigration: from black's equation to modern TCAD models. Microelectron Reliab 50:775–789
80. Park YH, Pasricha S, Kurdahi F, Dutt N (2007) System level power estimation methodology with h.264 decoder prediction IP case study. In: 25th international conference on computer design, ICCD 2007, pp 601–608. doi:10.1109/ICCD.2007.4601959
81. Park Y, Pasricha S, Kurdahi F, Dutt N (2008) Methodology for multi-granularity embedded processor power model generation for an ESL design flow. IEEE/ACM CODES+ISSS
82. Pasricha S, Dutt N, Ben-Romdhane M (2004) Extending the transaction level modeling approach for fast communication architecture exploration. In: Proceedings of 41st design automation conference, New York, pp 113–118
83. Pasricha S, Dutt N, Ben-Romdhane M (2006) Constraint-driven bus matrix synthesis for MPSoC. In: Asia and South Pacific conference on design automation, pp 6. doi:10.1109/ASP-DAC.2006.1594641
84. Pasricha S, Park YH, Kurdahi F, Dutt N (2006) System-level power-performance trade-offs in bus matrix communication architecture synthesis. In: Proceedings of the 4th international conference hardware/Software codesign and system synthesis, CODES + ISSS '06, pp 300–305. doi:10.1145/1176254.1176327
85. Pinto A, Carloni L, Sangiovanni-Vincentelli A (2003) Efficient synthesis of networks on chip. In: Proceedings of 21st international conference on computer design, pp 146–150. doi:10.1109/ICCD.2003.1240887
86. Powell MD, Biswas A, Emer JS, Mukherjee S, Sheikh B, Yardi S (2009) Camp: a technique to estimate per-structure power at run-time using a few simple parameters. In: IEEE 15th international symposium on high performance computer architecture (HPCA'09). IEEE
87. Rabaey JM (1996) Digital integrated circuits: a design perspective. Prentice-Hall, Inc., Upper Saddle River
88. Rao R, Vrudhula S, Chakrabarti C (2007) Throughput of multi-core processors under thermal constraints. In: Proceedings of the 2007 international symposium on low power electronics and design. ACM
89. Ravi S, Raghunathan A, Chakradhar S (2003) Efficient RTL power estimation for large designs. In: Proceedings of 16th international conference on VLSI design, pp 431–439. doi:10.1109/ICVD.2003.1183173
90. Rinard M (2006) Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In: Proceedings of the 20th annual international conference on supercomputing, ICS '06. ACM, New York, pp 324–334. doi:10.1145/1183401.1183447.

91. Rinard M, Hoffmann H, Misailovic S, Sidiroglou S (2010) Patterns and statistical analysis for understanding reduced resource computing. In: Proceedings of the ACM international conference on object oriented programming systems languages and applications, OOPSLA '10. ACM, New York, pp 806–821. doi:10.1145/1869459.1869525
92. Rodriguez S, Jacob B (2006) Energy/power breakdown of pipelined nanometer caches (90 nm/65 nm/45 nm/32 nm). In: Proceedings of the 2006 international symposium on low power electronics and design, ISLPED'06, pp 25–30. doi:10.1109/LPE.2006.4271802
93. Rohou E, Smith M (1999) Dynamically managing processor temperature and power. In: 2nd workshop on feedback-directed optimization
94. Sami M, Sciuto D, Silvano C, Zaccaria V (2000) Instruction-level power estimation for embedded VLIW cores. In: Proceedings of the eighth international workshop on hardware/-software codesign, CODES 2000, San Diego, pp 34–38
95. Sampson A, Dietl W, Fortuna E, Gnanapragasam D, Ceze L, Grossman D (2011) Enerj: approximate data types for safe and general low-power computation. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation, PLDI '11. ACM, New York, pp 164–174. doi:10.1145/1993498.1993518
96. Sanchez H, Philip R, Alvarez J, Gerosa G (1997) A CMOS temperature sensor for PowerPC RISC microprocessors. In: Proceedings of the symposium on VLSI circuits. IEEE, pp 13–14
97. Sarrigeorgidis K, Rabaey J (2004) Ultra low power cordic processor for wireless communication algorithms. J VLSI Signal Process Syst Signal Image Video Technol 38(2):115–130. doi:10.1023/B:VLSI.0000040424.11334.34
98. Sarta D, Trifone D, Ascia G (1999) A data dependent approach to instruction level power estimation. In: Proceedings of IEEE Alessandro volta memorial workshop on low-power design, pp 182–190. doi:10.1109/LPD.1999.750419
99. Shanbhag N(2002) Reliable and energy-efficient digital signal processing. In: Proceedings of 39th design automation conference, pp 830–835. doi:10.1109/DAC.2002.1012737
100. Shatzkes M, Lloyd JR (1986) A model for conductor failure considering diffusion concurrently with electromigration resulting in a current exponent of 2. J Appl Phys 59, 3890–3893
101. Shin JY, Kurdahi F, Dutt N (2015) Cooperative on-chip temperature estimationusing multiple virtual sensors. IEEE Embed Syst Lett 7(2):37–40. doi:10.1109/LES.2015.2400992
102. Skadron K, Stan MR, Huang W, Velusamy S, Sankaranarayanan K, Tarjan D (2003) Temperature-aware computer systems: opportunities and challenges. IEEE Micro 23(6): 52–61
103. Skadron K, Stan M, Sankaranarayanan K, Huang W, Velusamy S, Tarjan D (2004) Temperature-aware microarchitecture: modeling and implementation. ACM Trans Archit Code Optim 1:94–125
104. Sloan J, Sartori J, Kumar R (2012) On software design for stochastic processors. In: Proceedings of the 49th annual design automation conference, DAC '12. ACM, New York, pp 918–923. doi:10.1145/2228360.2228524
105. Uht A (2004) Going beyond worst-case specs with teatime. Computer 37(3):51–56. doi:10.1109/MC.2004.1274004
106. Wan L, Chen D (2010) Analysis of circuit dynamic behavior with timed ternary decision diagram. In: 2010 IEEE/ACM international conference on computer-aided design (ICCAD), pp 516–523. doi:10.1109/ICCAD.2010.5653852
107. Wang H, Tan S, Swarup S, Liu X (2013) A power-driven thermal sensor placement algorithm for dynamic thermal management. In: Design, automation & test in Europe conference & exhibition (DATE'13). IEEE
108. Wu W, Jin L, Yang J, Liu P, Tan S (2006) A systematic method for functional unit power estimation in mircoprocessors. In: 2006 43rd ACM/IEEE on design automation conference. IEEE
109. Ye W, Vijaykrishnan N, Kandemir M, Irwin M (2000) The design and use of simplepower: a cycle-accurate energy estimation tool. In: Proceedings of 2000 design automation conference, pp 340–345. doi:10.1109/DAC.2000.855333

110. Zaynoun S, Khairy M, Eltawil A, Kurdahi F, Khajeh A (2012) Fast error aware model for arithmetic and logic circuits. In: 2012 IEEE 30th international conference on computer design (ICCD), pp 322–328. doi:10.1109/ICCD.2012.6378659
111. Zhang Y, Li Y, Li X, Yao SC (2013) Strip-and-zone micro-channel liquid cooling of integrated circuits chips with non-uniform power distributions. In: ASME 2013 heat transfer summer conference
112. Zhang Y, Shi B, Srivastava A (2010) A statistical framework for designing on-chip thermal sensing infrastructure in nano-scale systems. IEEE Trans Very Large Scale Integration (VLSI) Syst 22(2):270–279

Part VIII
Codesign Tools and Environment

MAPS: A Software Development Environment for Embedded Multicore Applications

28

Rainer Leupers, Miguel Angel Aguilar, Juan Fernando Eusse, Jeronimo Castrillon, and Weihua Sheng

Abstract

The use of heterogeneous Multi-Processor System-on-Chip (MPSoC) is a widely accepted solution to address the increasing demands on high performance and energy efficiency for modern embedded devices. To enable the full potential of these platforms, new tools are needed to tackle the programming complexity of MPSoCs, while allowing for high productivity. This chapter discusses the *MPSoC Application Programming Studio (MAPS)*, a framework that provides facilities for expressing parallelism and tool flows for parallelization, mapping/scheduling, and code generation for heterogeneous MPSoCs. Two case studies of the use of MAPS in commercial environments are presented. This chapter closes by discussing early experiences of transferring the MAPS technology into Silexica GmbH, a start-up company that provides multi-core programming tools.

Acronyms

API	Application Programming Interface
ASAP	As Soon as Possible
AST	Abstract Syntax Tree
CG	Call Graph
CPN	C for Process Networks

R. Leupers (✉) • M.A. Aguilar • J.F. Esusse
Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Aachen, Germany
e-mail: leupers@ice.rwth-aachen.de; aguilar@ice.rwth-aachen.de; eusse@ice.rwth-aachen.de

J. Castrillon
Center for Advancing Electronics Dresden, TU Dresden, Dresden, Germany
e-mail: jeronimo.castrillon@tu-dresden.de

W. Sheng
Silexica GmbH, Köln, Germany
e-mail: sheng@silexica.com

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_2

DCG	Dynamic Call Graph
DFG	Dependence-Flow Graph
DLP	Data-Level Parallelism
DSP	Digital Signal Processor
FIFO	First-In First-Out
ICT	Information and Communications Technology
IP	Intellectual Property
IR	Intermediate Representation
ISA	Instruction-Set Architecture
KPN	Kahn Process Network
LLVM	Low-Level Virtual Machine
MAPS	MPSoC Application Programming Studio
MIMO	Multiple Input Multiple Output
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
NRE	Non-Recurring Engineering
PE	Processing Element
PLP	Pipeline-Level Parallelism
PNG	Portable Network Graphics
RAW	Read-After-Write
RLD	Run Length Decoding
SDF	Synchronous Data Flow
SESE	Single-Entry Single-Exit
TLP	Task-Level Parallelism
TRM	Trace Replay Module
WAR	Write-After-Read
WAW	Write-After-Write

Contents

28.1 Introduction

MPSoC Application Programming Studio (MAPS) is a package of advanced embedded multi-core programming technologies and tools with dedicated support for heterogeneous target platforms. Early research on MAPS started around a decade ago, when it became obvious that multi-core architectures would be the clear winner in almost all ICT applications, due to their power efficiency and scalability advantages. Early multi-core designs, such as the Texas Instruments C80 video DSP, already indicated the trend and possibilities but were initially not well accepted by the market. Today, however, the multi-core trend seems irreversible: smartphones do not sell without multi-core application processors, multi-core sensors assist the human driver in a car, and much of the Internet traffic is routed over multi-core wireless base stations. Multi-core-based devices with hundreds or even thousands of programmable processor cores will soon be a commodity.

The major issue with the multi-core trend in hardware platforms is that software programming technologies do not keep pace easily. Human programmers clearly prefer sequential, not parallel, programming languages and models. Moreover, the industry employs a huge amount of certified sequential legacy code, which cannot be ported into new parallel languages at any reasonable time and effort. As a consequence, software implementation for multi-core systems became a tedious and error-prone task, demanding for major innovations in multi-core-aware software compiler technology. This is not the first *software productivity crisis* of this kind: For instance, when domain-specific processors like DSPs became popular in the 1990s, virtually no usable compiler (and not even simulator) support was available at the beginning. Thus, DSPs had to be programmed "the hard way," i.e., in assembly language or using other low-level tools. Major R&D efforts from the research community and the semiconductor makers afterward helped to make DSP programming much more efficient via high-level languages like C/C++. Vice versa, processors without appropriate programming tools support quickly disappeared from the semiconductor market or even never fully made it there. The current situation with embedded multi-core systems is somewhat similar. There is a strong technology push from the semiconductor/IP segment, but the market adoption could be much faster if there were more convenient programming tools. Taking into account the 100+ Million US$ NRE cost for typical System-on-Chip designs today, software development tools as an afterthought means a significant risk for the semiconductor/IP industry.

Presently, three groups of actors aim at mitigating the "multi-core software development crisis":

1. The **academic research community** is working on novel parallelizing compiler technologies, suitable for embedded multi-core applications, while meeting specific constraints, e.g., real-time requirements and heterogeneous architectures. Moreover, staying with sequential programming is clearly no viable option in the long term, and new parallel, yet usable, programming languages (or multi-core-friendly subsets/supersets of existing languages) are receiving interest.
2. Various **start-up companies** are commercializing research results and are providing innovative tools and services to facilitate at least certain sub-tasks in multi-core programming, e.g., parallel performance estimation or parallelism detection.
3. The **semiconductor/IP industry**, in an effort to provide at least some "ad-hoc" level of programming convenience to their customers, typically ships products with native C/C++ compilers for the individual processor cores, variants of parallel programming models, such as OpenMP, or customized "semi-standard" programming APIs.

Currently, there is no clear winning programming standard in sight that would play the role of the well-proven and widespread C/C++ languages at the beginning of the multi-core era. Maybe multi-core programming will continue to be performed with a variety of domain-specific tools. More research and practical experience with industrial applications and user acceptance are clearly required. MAPS is aimed at contributing to this development by exploring new avenues in programming, yet taking practical issues into account. Like many similar projects, MAPS initially focused solely on parallelism analysis and partitioning of sequential C code [12]. Later it turned out that code partitioning is actually just one aspect in multi-core programming; hence, the technology was significantly enhanced by parallel software task mapping and scheduling, specific language support, target architecture modeling formalisms, profiling and performance estimation techniques, and integration into legacy software stacks. Figure 28.1 shows an overview of the MAPS framework. The following sections of this chapter provide details on these aspects:

- Section 28.2 analyzes programming model requirements for typical embedded application domains and describes *CPN*, a domain-specific C language extension for steaming-oriented applications.
- Section 28.3 briefly outlines the target platform modeling facilities in MAPS.
- Section 28.4 describes a MAPS-based sequential programming flow, including a variety of features for parallelism pattern detection and exploitation.
- Section 28.5 shows a MAPS-based parallel programming flow, including CPN-based task mapping and parallel software performance estimation techniques.
- Section 28.6 shows a MAPS-based code generation flow, which translates the CPN specification into plain C code, which is then compiled with the toolchain of the target platform.
- Section 28.7 provides results of two case studies, where MAPS has been applied for code optimization in a typical Android software stack set-up as well as for code parallelization for an advanced multi-core DSP platform.

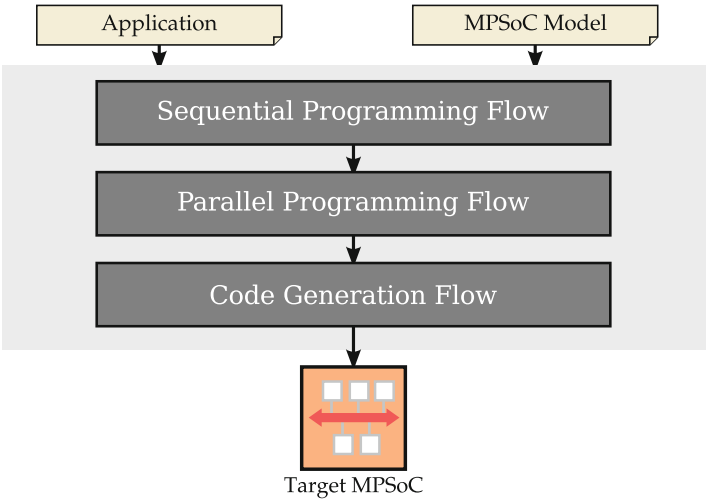


Fig. 28.1 MAPS framework overview

Finally, Sect. 28.8 provides an early industrial perspective. Following significant industrial interest in the MAPS technology (e.g., in the form of user workshops or bilateral academia-industry projects), Aachen-based Silexica GmbH obtained an exclusive license of the MAPS technology in 2014 in order to provide robust software tool products for a fast-growing embedded developers' market. Some lessons learned from this technology transfer enable important feedback into further, fundamental and applied, MAPS research activities.

28.2 Application Software Programming

MAPS focuses on streaming models typically found in wireless, multimedia, and signal processing applications. The input application described in Fig. 28.1 is based on these models. A short description of the streaming programming model used within MAPS is presented in this section. After a thorough study of several different approaches utilized to practically realize such models, a language-extension-based approach was chosen because of its convenience and applicability. An overview of the proposed language extensions is discussed throughout the section.

28.2.1 Streaming Models of Computation

Streaming (or data flow) models have gained acceptance in the embedded community (e.g., wireless and multimedia), since applications in this domain typically follow a streaming-based computation approach [33]. Kahn Process Networks

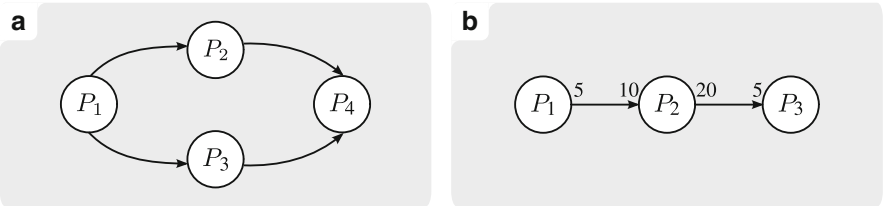


Fig. 28.2 Process network examples: (**a**) KPN example. (**b**) SDF example. Numbers indicate how many tokens are consumed or produced per iteration

(KPNs) [21] are a widespread Model of Computation (MoC), used to describe, e.g., signal processing applications. A KPN is represented as a directed graph. Nodes within the graph represent autonomous processes that perform computations, and edges between nodes represent unbounded unidirectional First-In First-Out (FIFO) channels that transmit data tokens. Reading from an empty FIFO channel results in the process being blocked until data to read becomes available. A KPN example composed of four processes is shown in Fig. 28.2a. The Synchronous Data Flow (SDF) model, a subset of KPN, has also been defined [25]. An SDF requires that the control flow of a process be an endless loop, which computes over predefined fixed channel accesses. Figure 28.2b presents an example of an SDF with three processes. For each iteration of the SDF shown in the figure, process $P2$ always consumes 10 tokens from $P1$ and produces 20 tokens for $P3$. The use of SDFs to specify signal processing algorithms is widespread, given the large amount of algorithms that strictly follow this model (e.g., signal sampling and filtering). KPNs and SDFs resemble the way humans think of parallel processing and are therefore perceived as *intuitive* ways to specify streaming programs.

28.2.2 C for Process Networks (CPN)

MAPS defines a lightweight C language extension called *C for Process Networks (CPN)* to capture streaming models. CPN allows to describe the semantics of process networks at a high level, by adding a small set of new keywords intended to specify processes and channels. This design enables retargetability toward embedded MPSoCs, where processing elements have different APIs and machine-specific low-level primitives. Moreover, CPN programs are inherently portable, which opens abundant opportunities for code transformations and optimizations. A common criticism is that C is not an ideal programming language for parallelism specification. Nonetheless, a compromise is needed considering the large legacy C code base and the language popularity in the embedded industry. Some basic elements of the CPN language are now introduced. The new keywords (prefixed with `__PN`) are used for channels, processes, and channel accesses of the supported streaming models.

Listing 1 CPN example code: channel declaration

```
1  typedef struct { int i; double d; } my_struct_t;
2
3  __PNchannel char B[3][3];
4  __PNchannel my_struct_t C;
5  __PNchannel int A = {1024, 2048, 4096}; /* Initial channel tokens */
```

Listing 2 CPN example code: KPN process template (run length decoding)

```
1  /* Run Length Decoding, e.g. 4A2B5C3D -> AAAABBCCCCCDDD */
2  __PNkpn RLD __PNin(int EncIn) __PNout(int DecOut)
3  {
4    int count, i;
5    while (1) {
6      __PNin(EncIn) /* read a token (# of appearances) from EncIn */
7        count = EncIn;
8      __PNin(EncIn) /* read a token (data itself) from EncIn */
9        for (i = 0; i < count; ++i) /* write data to DecOut */
10         __PNout(DecOut)
11         {
12           DecOut = EncIn;
13         }
14   }
15 }
```

28.2.2.1 Channels

Channel declarations in CPN are done using the `__PNchannel` keyword, which is similar as declaring a global variable in C. Channel declaration examples are presented in Listing 1. Elementary C types, such as `int`, `char`, `float` and enumerations are valid channel types. Structures, unions, and arrays of valid channel types are also valid. By default a channel is set to be empty at the beginning of program execution. If initial channel tokens are desired (i.e., to avoid deadlocks), the specification of initialization lists in the channel declaration (e.g., channel `A` at Line 5 in Listing 1) is also supported by CPN.

28.2.2.2 Processes

The concept of *process templates* is used in CPN to allow code reuse, similar to a C function. A process template describes the functionality of a process, the channels accessed by this process, and their access type (read or write). Processes are then created as *instances* of process templates. The readability and conciseness of the CPN code is improved using this approach, when multiple processes in a network perform identical computations.

Run Length Decoding (RLD) is used as an example to illustrate the concept of process templates. RLD is a technique used in fax machines for data decompression, where the input data is encoded into a stream of pairs composed of the number of appearances and the data element itself. For example, the original string `AAAABBCCCCCDDD` is compressed into `4A2B5C3D`. Listing 2 shows a possible implementation of RLD in CPN. The KPN process template (`__PNkpn`)

Listing 3 CPN example code: process instantiation

```
1  __PNchannel int dec1_in = {4, 'A', 2, 'B', 5, 'C', 3, 'D'};
2  __PNchannel int dec2_in = {3, 'E', 5, 'F', 4, 'G', 2, 'H'};
3  __PNchannel int dec1_out, dec2_out;
4
5  __PNprocess src = Src __PNout(dec1_in, dec2_in);
6  __PNprocess dec1 = RLD __PNin(dec1_in) __PNout(dec1_out);
7  __PNprocess dec2 = RLD __PNin(dec2_in) __PNout(dec2_out);
8  __PNprocess add = Add __PNin(dec1_out, dec2_out);
```

with identifier `RLD` is first declared (Line 2). The process reads integers from its input channel `EncIn` and writes integers to the output channel `DecOut`. This is indicated by keywords `__PNin` and `__PNout`, respectively. Arbitrary C code can be embedded in the body of a KPN process template, which allows the accesses to input and output channels at any point of the computation. Listing 3 shows how process instances are created from the process templates and how channels are connected to them. This CPN code corresponds to the KPN topology shown in Fig. 28.2a.

28.3 MPSoC Target Architecture Modeling

A Multi-Processor System-on-Chip (MPSoC) is composed of processing elements of multiple types, storage elements, interconnects, and peripherals. As described in Fig. 28.1, MAPS takes as one of its inputs an MPSoC model. This model is required to enable the analyses performed by the sequential and parallel flows presented in Sects. 28.4 and 28.5, respectively. The architecture model provides a simplified view of the target MPSoC, which describes it in terms of processing elements and communication primitives. Before formally defining the MPSoC model, some preliminary notions are required for clarity [8].

A processor type (PT) stands for a processor architecture that is instantiated in an MPSoC, possibly multiple times. The PT specifies a cost model for every operation in its *Instruction-Set Architecture (ISA)* and also attributes that provide additional information about the run-time system, such as context switch time and available scheduling policies. The set of all processor types in an MPSoC is denoted as $\mathscr{PT} = \{PT_1, \ldots, PT_{N_{PT}}\}$. A processing element ($PE$) is an instance of a given processor type, and it inherits its cost model and attributes. The set of all PEs can be defined as the disjoint union of all the processor type instances $\mathscr{PE} = \sqcup_{v \in \mathscr{PT}} \mathscr{PE}^v$, where $\mathscr{PE}^v = \{PE_1^v, \ldots, PE_{N_{PE}}^v\}$ is the set that contains all the instances for a particular type ($PT_v \in \mathscr{PT}$).

Similarly, a communication primitive (CP) represents software APIs available to allow communication between tasks within an application. It is defined as the triple $CP = (PE_i, PE_j, \mathscr{CM}^{CP})$, which expresses how the task in PE_i communicates with the task in PE_j, where $i = j$ is allowed. $\mathscr{CM}^{CP}$ is the cost model of the communication primitive that consists of functions that associate an amount of

communicated data with a numerical value. An example of a function contained in $\mathscr{CM}^{CP}$, is denoted as $\zeta^{CP} : \mathbb{N} \to \mathbb{N}$, such that $\zeta^{CP}(b)$ returns the cycle count of transmitting b bytes over CP. The set of all CPs in the MPSoC is denoted as $\mathscr{CP}$. The $MPSoC$ model is then defined as follows:

Definition 1 (MPSoC Model). The model of the target MPSoC is a multigraph $MPSoC = (\mathscr{PE}, \mathscr{E})$, where $\mathscr{E}$ is a multiset over $\mathscr{PE} \times \mathscr{PE}$ that contains an element $e_{ij} = (PE_i, PE_j)$ for every $CP \in \mathscr{CP}$. For convenience, let CP_{ij} denote the communication primitive associated with e_{ij}.

28.4 Sequential Programming Flow

This section describes the details of the tool flow that is used to identify multiple forms of parallelism within sequential C code in embedded applications. The outcome of this parallelization flow are hints that guide the developer in the process of deriving a CPN representation of the input application.

28.4.1 Tool Flow Overview

An overview of the sequential programming flow (or parallelization flow) is shown in Fig. 28.3. Its inputs are the sequential C code of the application, a model of the MPSoC, and constraints provided by the developer. A *program model* is constructed based on *static* information gathered while performing source code compilation, and *dynamic* information gathered through application profiling. Afterward, the program model is analyzed by algorithms to detect and classify multiple forms of parallelism.

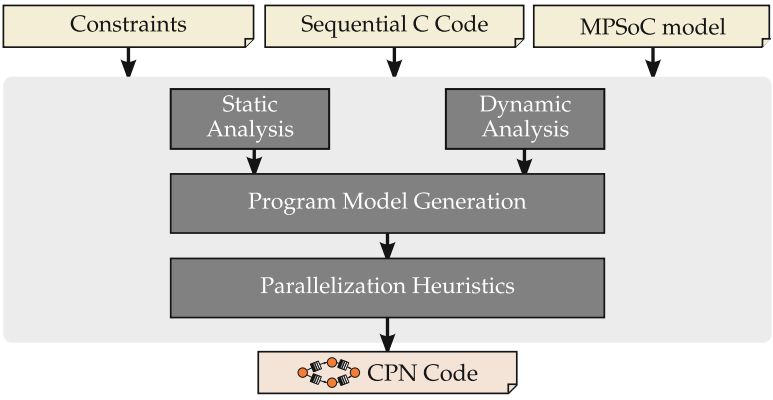


Fig. 28.3 Sequential programming flow

Finally, information in the form of source-level annotations is presented to the developer to guide the process of deriving a CPN representation of the application. The following sections describe the relevant details of the tool flow.

28.4.2 Program Model

The main data structure of the sequential programming flow is the *Program Model*. This model describes the application in terms of performance information, and a set of graphs that contain control and data dependency relationships between the statements within functions in the application. In the following sections, the program model is described in detail.

28.4.2.1 Performance Information

A key element for a profitable parallelization is performance information. It is used for two purposes: identifying computationally intensive functions (termed here *parallelization candidates*) and performing a cost-benefit analysis to evaluate the potential speedup of parallelizing a given candidate. The sequential flow relies on a microarchitecture-aware cost table model to derive a performance estimate of the application, which is enabled by the MPSoC model and the dynamic information.

28.4.2.2 Dynamic Call Graph

In the context of parallelism extraction, a complete view of the application is required [8]. This is enabled by a *Call Graph (CG)*, which is composed of a set of nodes that represent functions and edges that represent calling relationships among functions. A *Dynamic Call Graph (DCG)* is used in the sequential programming flow. This is a special type of CG, which only contains functions that were executed in a given profiling run. In a DCG, each function node is annotated with its *self cost*, which considers only actual computations within the function. The *total cost* of the function is annotated as well, which also considers the computations performed by called functions. The DCG edges also contain information about the number of times and the source code location, where each call took place. Building an application DCG enables the detection of computationally intensive functions (i.e., *parallelization candidates*), which are extracted by applying a user-defined threshold. This threshold sets the minimum percentage of the total application workload that a given function must have in order to be considered as a *parallelization candidate*.

28.4.2.3 Dependence-Flow Graph

To enable parallelism extraction, each parallelization candidate is represented as a *Dependence-Flow Graph (DFG)* [20]. A DFG not only combines control and data dependencies but also exploits the program structure by exposing *Single-Entry Single-Exit (SESE)* regions. Before introducing the concepts of a DFG, some definitions are needed for clarity. An intermediate representation ($\mathscr{IR}^A$) of an

application A is a pair $\mathscr{IR}^A = (S^A_{stmt}, S^A_f)$, where $S^A_{stmt} = \{s_1, \ldots, s_n\}$ is the set of all IR statements generated by a C compilation. $S^A_f = \{f_1, \ldots, f_m\}$ is the set of all functions defined in the application. Each function $f \in S^A_f$ maps to a subset of statements $S^f_{stmt} \subset S^A_{stmt}$ and can be also expressed as a control-/data-flow graph $CDFG^{f_j} = (S^{f_j}_{stmt}, E^{f_j})$, where the edge set E^{f_j} contains the control and data dependencies for the function, as defined in the following.

Let δ^c be a control dependency between two statements s_u and s_v. It represents the case where a condition present in statement s_u evaluates in a way that allows the execution of statement s_v. The set of all control dependencies in a function f_j is denoted as $E^{f_j}_c$. Similarly, let δ^d be a data dependency between two statements s_u and s_v. It represents the case where both statements access the same variable or processor resource. The set of all data dependencies in a function f_j is denoted as $E^{f_j}_d$. There are three different types of data dependencies, namely, *Read-After-Write (RAW)*, *Write-After-Write (WAW)*, and *Write-After-Read (WAR)*.

A Single-Entry Single-Exit (SESE) is a connected subgraph, such that there exists a unique incoming control edge from outside the region and a unique outgoing control edge to outside the region. The set of all the SESE regions of a function f_j is denoted here as $R^{f_j} = (r^{f_j}_1, \ldots, r^{f_j}_{N_R})$. Due to their structured control flow, SESE regions are a convenient unit for parallelism detection. Moreover, SESE regions can be identified based on the language construct that they represent (e.g., *if-then-else*, or *loops*) as the examples in Fig. 28.4 show. Only data dependencies relevant for the regions are considered within the SESE regions. In Fig. 28.4a the dependency

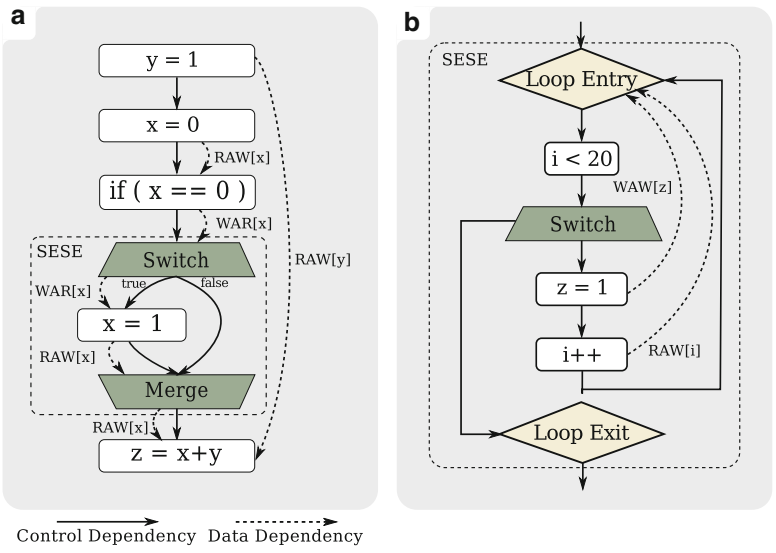


Fig. 28.4 Dependence-Flow Graph (DFG) examples: (**a**) if-then-else and (**b**) loop

on variable x is redirected inside the region as there is an operation that utilizes it, while the dependency on y bypasses the region. Based on the previous notions it is now possible to define a DFG.

Definition 2 (Dependence-Flow Graph). A Dependence-Flow Graph (DFG) of a function $f_j \in F$ is a directed multigraph $DFG^{f_j} = (S_{stmt}^{f_j}, E_c^{f_j} \cup E_d^{f_j}, R^{f_j})$, where $S_{stmt}^{f_j}$ is the set of function statements, $E_c^{f_j} \cup E_d^{f_j}$ is the union of control and data dependency sets, and R^{f_j} is the set of SESE regions. The set of all DFGs given an application execution is defined as $\mathscr{DFG}^F$.

There are multiple types of nodes within a DFG. *Statement* nodes represent operations and function calls, which are augmented with run-time execution counts. *Switch* nodes are conditional jumps to multiple targets. In contrast to the original definition of DFG in [20], additional *loop entry* and *exit* nodes are defined, which enclose and redirect data dependencies within loop regions. This is illustrated in Fig. 28.4b. Additionally, *control* edges are annotated with the direction of the control flow (forward or backward) and their execution count obtained from dynamic information. Finally, *data* edges are annotated with detailed information about the carried dependency.

28.4.2.4 Loop Analysis

Loop analysis is an important step within the sequential programming flow, as loops are typically one of the most interesting constructs for parallelization. Information extracted during this analysis is annotated on the loop entry nodes, and is later used during the parallelism identification phase. This information includes:

- *Induction variables*: these are typically used to control the iterations of loops and they are modified as $i = i \oplus exp$, where i is the induction variable, $\oplus$ is the operator and exp can be either loop-invariant or derived from another induction variable.
- *Private variables*: these are used only within the same iteration in which they are defined [22]. Since they do not have to be exposed to other processes at run time, it can be safely assumed that the loop-carried dependencies of these variables do not prevent parallelization.
- *Reduction variables*: reduction operations are a common pattern found in many loops [22]. A reduction variable accumulates values across iterations of a loop. They have the form of $r = r \oplus exp$, where r is the reduction variable, which is not defined or used anywhere else in the loop, $\oplus$ is a commutative and associative operator, and exp is typically a loop-variant expression.
- *Loop-carried dependencies*: data dependencies across loop iterations that are not related to induction, reduction, or private variables are annotated as actual loop-carried dependencies. They can be identified by analyzing backward data dependencies in the SESE that represents each loop.

- *Profiling information*: details about the average trip count and the number of times a loop was entered in a given profiling run, are also annotated on the loop entry node. This information is relevant to perform a cost-benefit analysis when a given loop is considered for parallelization.

28.4.2.5 Program Model Definition and Construction

Using the aforementioned notions, it is possible to formally define the *program model*, which is the basis for the parallelism discovery process in MAPS.

Definition 3 (Program Model). A program model is defined as the tuple $\mathscr{PM} = (\mathscr{DFG}^F, DCG)$, where $\mathscr{DFG}^F$ is the set of Dependence-Flow Graphs (DFGs) and DCG is the Dynamic Call Graph (DCG).

The construction of the $\mathscr{PM}$ starts by creating statement nodes and control flow edges for the executed functions at the compiler IR level. Information about the definition and use of variables is annotated on the nodes by using the compiler IR (i.e., static) and profiling (i.e., dynamic) information. SESE regions are then extracted, and the nodes that delimit them are inserted (i.e., Switch, Merge, Loop Entry and Exit). Finally, data dependency edges are created by using variable access information. Once a DFG is built for each executed function, the DCG is constructed taking into account the performance estimation.

In contrast to purely static analyses, approaches that involve dynamic information are not *fully safe*. The reason is that dynamic analysis cannot guarantee that all possible data dependencies in an application are discovered for a given input. In order to mitigate this problem, code coverage analysis is performed to provide the developer with feedback about the code parts that were actually executed.

28.4.3 Parallelism Identification

In this section, the heuristics used to identify multiple forms of parallelism within sequential C code are presented. The heuristics use the DFGs and the performance information contained in the program model to perform the analyses at the C statement level of granularity. This provides great flexibility and allows a direct correlation between the extracted processes and the source code. The final results are source-level parallelization hints that guide the developers in the process of transforming the input C code into a CPN representation. The heuristics described in this section were introduced in [8] and later extended in [2,3].

28.4.3.1 Task-Level Parallelism (TLP)

In TLP, a computation is divided into different processes that operate concurrently as Fig. 28.5a shows. Here, Processes 1 and 2 are able to run in parallel as the only dependency on variable `x` appears early in Process 1, thus allowing Process 2 to start its execution. Algorithm 1 shows the TLP extraction heuristic. As an

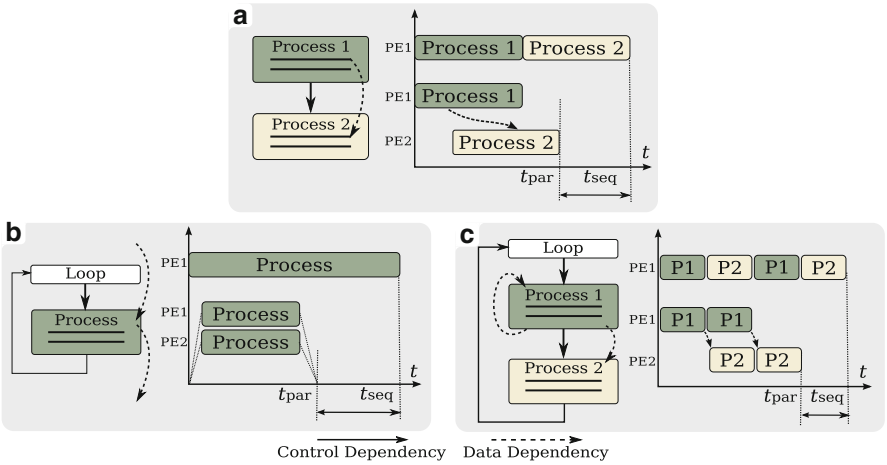


Fig. 28.5 Forms of parallelism: (**a**) task-level parallelism, (**b**) data-level parallelism, and (**c**) pipeline-level parallelism

Algorithm 1 Task-level parallelism extraction

```
1:  procedure ExtractTLP(𝒫𝓜, MPSoC, w_thld, η_thld)
2:      DCG ← GetDCG(𝒫𝓜), PC ← GetParallelizationCandidates(DCG, w_thld)
3:      for f_j ∈ PC do
4:          DFG^{f_j} ← GetDFG(f_j)
5:          DFG_c^{f_j} ← CollapseSubregions(DFG^{f_j})
6:          P ← GetFirstPartition(DFG_c^{f_j})
7:          while GetSuccessor(P) ≠ ∅ do
8:              P' ← GetSuccessor(P)
9:              t_seq ← GetSeqTime(P, P', MPSoC)
10:             t_par ← ASAP(P, P', MPSoC)
11:             η ← t_seq/(2 · t_par)
12:             if (η < η_thld) then
13:                 P ← P ∪ P'
14:             end if
15:         end while
16:     end for
17: end procedure
```

input, it takes the program model ($\mathscr{PM}$) and the user-defined threshold (w_{thld}), which specifies the minimum workload to consider a function as a parallelization candidate. Then all the structures such as if-then-else blocks and loops are collapsed as single partitions to get a linear control flow (Line 5). Afterward, two consecutive partitions in the control flow are jointly analyzed. The parallel execution time is computed by performing an ASAP scheduling of both partitions. In case that the parallel efficiency $\eta = t_{seq}/(2 \cdot t_{par})$, is below a user-defined threshold η_{thld}, the partitions are merged (Lines 12 and 13).

28.4.3.2 Data-Level Parallelism (DLP)

This form of parallelism is typically found in scientific and multimedia applications. It is exploited by splitting the iteration space of a loop into multiple workers, provided that there are no loop-carried dependencies. Figure 28.5b shows an example of a loop with DLP. Algorithm 2 shows the heuristic to extract DLP. It takes as inputs the program model ($\mathscr{PM}$), the model of the target platform ($MPSoC$), and the user-defined threshold (w_{thld}) for parallelization candidates. For each parallelization candidate ($f_i \in PC$) the set of regions that represent loops is extracted. Then for each region a loop annotation LA^{r_j} is obtained (Line 7), containing relevant information as described in Sect. 28.4.2.4. A loop has to meet the following preconditions to be considered for DLP (Line 8): (i) there is only one induction variable and thus one single iteration space, (ii) there are no loop-carried dependencies, and (iii) the ratio between the average trip count and the number of times the loop was entered is bigger than a given threshold. If these preconditions are met, the algorithm iterates until the parallel efficiency (η) is below a user-defined threshold (η_{thld}) or the number of workers is equal to the number of cores in the target MPSoC (N_{PE}). The execution time of the parallelized code is modeled as $t_{par} = t_{seq}/workers + t_{comm}(workers, E_{io})$, where $workers$ is the number of utilized cores, E_{io} are the input and output data edges of the loop, and t_{comm} is the communication overhead. Here source-level annotations include details, such as the expected speedup and the list of induction, private, and reduction variables. In case that DLP cannot be exploited, the reasons are also presented to the developer.

Algorithm 2 Data-level parallelism extraction

```
1:  procedure ExtractDLP(𝒫𝓜, MPSoC, w_thld, η_thld)
2:      DCG ← GetDCG(𝒫𝓜), PC ← GetParallelizationCandidates(DCG,w_thld)
3:      N_PE ← GetNumCores(MPSoC)
4:      for f_j ∈ PC do
5:          DFG^{f_j} ← GetDFG(f_j), R_L^{f_j} ← GetLoopRegions(DFG^{f_j})
6:          for r_i ∈ R_L^{f_j} do
7:              LA^{r_i} ← GetLoopAnnotations(r_i)
8:              if HasValidPreconditions(LA^{r_i}) then
9:                  E_io ← GetIODataEdges(r_i)
10:                 workers ← 0
11:                 while (η > η_thld) ∨ (workers ≤ N_PE) do
12:                     t_seq ← GetExcTime(r_i, MPSoC)
13:                     t_par ← t_seq/workers + t_comm(workers, E_io)
14:                     η ← t_seq/(workers · t_par)
15:                     workers ← workers + 1
16:                 end while
17:             end if
18:         end for
19:     end for
20: end procedure
```

Algorithm 3 Pipeline-level parallelism extraction

```
1:  procedure ExtractPLP(𝒫𝓜, MPSoC, w_thld)
2:      DCG ← GetDCG(𝒫𝓜), PC ← GetParallelizationCandidates(DCG, w_thld)
3:      for f_j ∈ PC do
4:          DFG^{f_j} ← GetDFG(f_j), R^{f_j} ← GetLoopRegions(DFG^{f_j})
5:          for r_i ∈ R^{f_j} do
6:              LA^{r_i} ← GetLoopAnnotations(r_i)
7:              r_i^c ← ColapseSubregions(r_i)
8:              N_part ← GetPartNumber(r_i^c)
9:              if HasValidPreconditions(LA^{r_i}) ∨ IsOutermost(r_i) ∨ (N_part > 2) then
10:                 (t_par, P_cfg) ← LinearPipeBalance(r_i^c, MPSoC)
11:                 N_stg ← GetNumStages(P_cfg)
12:                 S_dom ← GetDomStage(P_cfg)
13:                 if (N_stg < GetNumCores(MPSoC) ∨ HasLoop(S_dom) then
14:                     (t'_par, P'_cfg) ← OptimizeDomStage(P_cfg)
15:                 end if
16:             end if
17:         end for
18:     end for
19: end procedure
```

28.4.3.3 Pipeline-Level Parallelism (PLP)

This is an important form of parallelism in embedded systems, as many applications in this domain perform stream-based computations [32]. In PLP, the computation of a loop body is broken into a sequence of steps (called *pipeline stages*) that can be executed in parallel as Fig. 28.5c shows. In this example the first stage communicates its result to the second. In contrast to DLP, PLP can be exploited in presence of loop-carried dependencies, which makes this pattern an interesting parallelization alternative. Algorithm 3 describes the process of PLP extraction. Similar to DLP, preconditions are evaluated for each loop region (Line 9). However, in PLP loop-carried dependencies are allowed and only the outermost loop is considered in first place. The heuristic continues by collapsing the sub-regions within the loop body as independent blocks to get a linear control flow. Afterward, the heuristic extracts the most balanced pipeline configuration (P_{cfg}) taking into account the MPSoC model (Line 10). A pipeline configuration P_{cfg} describes the number of pipeline stages and the mapping between the lines of code and the pipeline stage to which they belong. However, the performance of the extracted pipeline configuration could be limited by a *dominating stage* (S_{dom}), which is the slowest one and could prevent achieving a well-balanced configuration. Therefore, the configuration could be further optimized (P'_{cfg}), by analyzing nested loops within the dominating stage. If the dominating stage contains a loop (Line 13), it is possible to try to extract either DLP or PLP from it. This corresponds to *stage replication* and *multi-level pipeline extraction*, respectively [34]. The procedure in Line 14 performs the aforementioned optimization recursively, until all the available cores in the MPSoC are used or the dominating stage cannot be further optimized.

28.5 Parallel Programming Flow

This section describes the tool flow to generate a mapping configuration from a KPN application onto a heterogeneous MPSoC. The input application is described in CPN, which may be obtained as a result of the sequential programming flow in Sect. 28.4.

28.5.1 Tool Flow Overview

An overview of the parallel programming flow is shown in Fig. 28.6. Apart from the application specification and the MPSoC model, the flow receives a configuration file and a non-functional specification in form of application and mapping constraints. The flow itself is divided into two phases, one for obtaining traces that represent the interaction among KPN processes (see Sect. 28.5.2) and the other for computing the actual mapping (see Sect. 28.5.4). These phases require means for estimating or measuring performance to steer the optimization processes, as discussed in Sect. 28.5.3. Before delving into the details, the extra input/output specifications needed by the flow are introduced first in the following.

28.5.1.1 Constraints and Configuration

The flow supports multiple types of application constraints. This includes *time*, *mapping*, and *platform* constraints. Time-related ones allow to set a throughput constraint on a KPN process and a latency constraint along a path in the KPN or to define processes that are triggered by timers. Mapping-related constraints enforce a solution for application elements (processes and channels). That is, they fix a processor to a process, or a communication primitive to a channel. Finally, platform constraints restrict the available resources visible to the application (e.g., a subset of

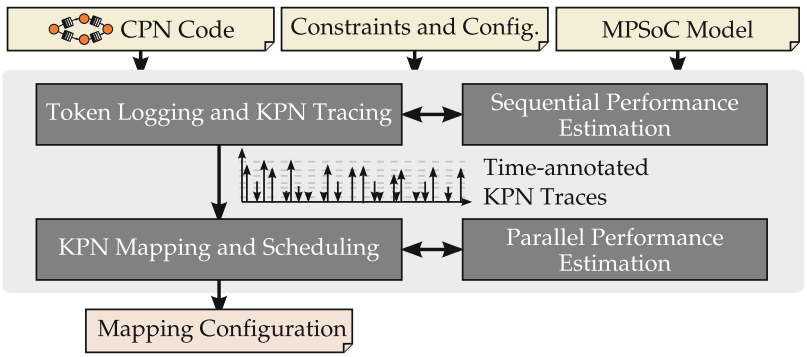


Fig. 28.6 Parallel programming flow

processing elements or lower memory resources). The configuration of the tool gives the user control over the optimization engine. This includes selecting the search algorithm for obtaining the mapping, as well as the algorithm parameters.

28.5.1.2 Mapping Configuration

As output, the mapping flow produces a so-called *mapping configuration*, which is used by the *cpn-cc* compiler to generate code accordingly, as discussed in Sect. 28.6. The configuration describes how processes are to be mapped to processors, which communication primitives are to be used for the logical channels, and the size of the KPN FIFO buffers.

28.5.2 Token Logging and KPN Tracing

In contrast to *static* data-flow programming models, where actor interactions are explicit in the specification, process interactions are hidden in a KPN specification. Depending on the internal control flow and the incoming data, inter-process communication may vary. This is illustrated in Fig. 28.7. The figure shows a sample KPN specification and the control flow of its processes in Fig. 28.7a, together with two hypothetical schedules in Fig. 28.7b, c. These schedules result from different paths along the control flow graph of process P_1. To characterize these interactions we use token logging and KPN tracing, as described in the following.

28.5.2.1 Token Logging

Token logging refers to the process of capturing the tokens that flow through all the channels of a KPN for a given execution. To achieve this, an instrumented *Pthreads* implementation of the CPN application is generated and linked against a custom FIFO library (one that writes the tokens to a binary file). Once the channel histories have been collected, it is possible to execute each process in isolation (e.g., using a simulator or a sequential performance estimator, as discussed in Sect. 28.5.3.1). Since KPNs are determinate, the paths along the control flow graphs will not change as a consequence of changing the schedule.

28.5.2.2 KPN Traces

As illustrated in Fig. 28.7, we are interested in capturing the events at which processes interact with each other. Knowing the occurrence time of the events allows the flow to reason about possible schedules (as shown in Fig. 28.7b, c). Channel accesses are important because they can potentially change the state of processes in the KPN application, e.g., a read from an empty channel would block the consumer process, whereas a read from a full channel would unblock the producer process. Apart from read and write accesses to channels, we also record the so-called time checkpoints. Time checkpoints appear in the trace as a consequence of a call to a function or a primitive that measures time (e.g., `waitForNextPeriod` in real-time applications). These events are used in the flow to check for violations of the time constraints specified by the programmer.

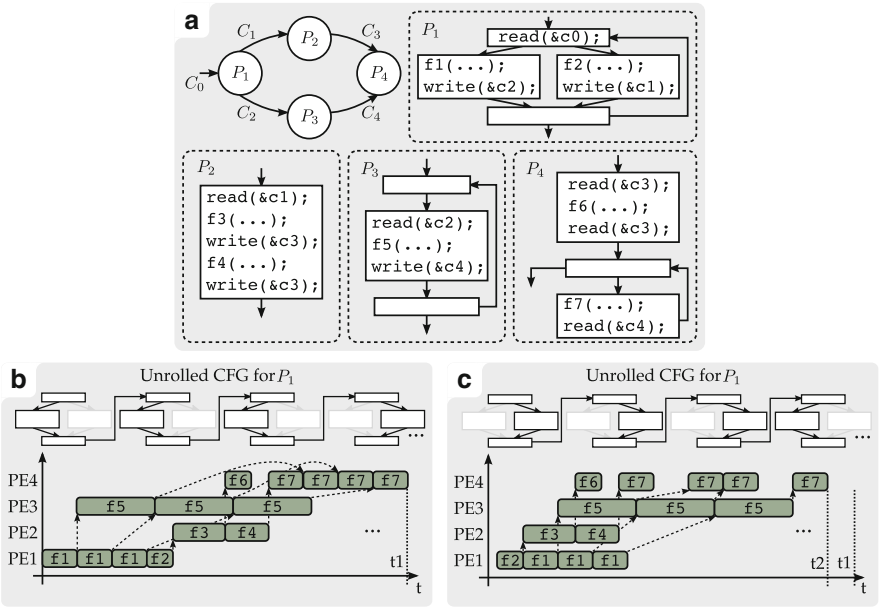


Fig. 28.7 Control flow and process interactions. (**a**) Sample KPN application. (**b**, **c**) Possible schedules

The execution of a process is then characterized by a sequence of events (trace), separated by arbitrary computations (segments). More formally, given a KPN graph $KPN = (\mathscr{P}, \mathscr{C})$, with $\mathscr{P}$ the set of processes and $\mathscr{C} \in \mathscr{P} \times \mathscr{P}$ the set of channels:

Definition 4 (Segment). A segment (S) of a process $P \in \mathscr{P}$ is a sequence of statements $S = (s_1, \ldots, s_n)$ that determine a path in its control flow graph, where s_1 follows immediately from a synchronization event (e.g., read, write, or iteration finish events) and s_n is the only statement in the sequence that generates a synchronization event.

Definition 5 (Process trace). A process trace T_P of a process $P \in \mathscr{P}$ is a sequence of segments $T_P = (S_1^P, \ldots, S_m^P)$ observed during the tracing process.

An application is represented as a set of traces, one for each process. This is a common representation for applications modeled as KPNs, or dynamic data-flow graphs [6, 9, 11, 14]. If different inputs are considered, then each process is characterized by multiple traces. In this case, traces can be analyzed to identify structure and derive mappings that are valid for sets of different traces [18]. In general, traces help to understand *how* processes interact, i.e., communicate. To really understand *when* the processes would interact, segments must be annotated with time estimates or measurements. This is the matter of the following section.

28.5.3 Trace Generation and Performance Estimation

Mapping and scheduling generation are directly influenced by the times in which different application processes interact. Such times depend on the processor type on which they will be executed according to the mapping decisions. Therefore, the aforementioned KPN traces must be generated for each of the processor types inside the target MPSoC (See Sect. 28.3), for each application process. Such traces are necessary for the success of the mapping and scheduling heuristics and are produced by sequential performance estimation.

Furthermore, a fast mechanism to predict parallel execution time is also required after mapping and scheduling have been performed. This is done to evaluate the profitability of the decisions and to discard poor candidates. This section starts by detailing the strategy through which KPN traces are generated for a parallel CPN application (i.e., sequential performance estimation). A brief explanation of the derivation of the execution times for an application that has been already mapped/scheduled is then presented.

28.5.3.1 Sequential Performance Estimation

The sequential performance estimation stage receives the application and the MPSoC model as an input and generates two outputs. The first is the aforementioned token-logging (*Pthreads*) version of the application, which is executed to generate an input/output token history that characterizes the activity in the FIFO channels. The second output consists of a standalone C application for each CPN process, which will read its corresponding input from the generated token history files. Each one of these standalone applications is then instrumented using CoEx [15], a Multi-Grained Source-Level Profiling tool. A trace containing function calls and basic block executions is generated while executing the instrumented binary, and it will be consumed as an input of the next stage in the estimation process.

The MPSoC model, the standalone applications, and their source-level execution traces are available at this stage. They are fed to a performance estimation engine [16], which calculates the clock cycles it takes to execute each basic block of a given standalone application, given a processor type. The engine behaves as a *pseudo-compiler back-end*, in the sense that it emulates one without performing code emission. In order to generate the cost for a given basic block, lowering, resource-constrained list scheduling, and code placement are performed. During *lowering*, the IR instructions of the input application that do not match to those supported by the processor model are transformed into simpler, supported ones. After lowering, each of the resulting basic block instructions is assigned to an execution step using list scheduling. Afterward, the position of basic blocks relative to each other (i.e., placement) is determined based on profiling information. This accounts for the fact that contiguously placed basic blocks do not need a branch instruction, improving the estimates.

After cost calculation, the estimation engine traverses the trace and accumulates the number of clock cycles elapsed between two channel accesses. This number

is then dumped together with the type of channel access, the channel ID and the source line where the access was performed, effectively creating a time-annotated KPN trace. The estimation is repeated for each CPN process, until all the KPN traces are available for each processor type of the MPSoC.

28.5.3.2 Parallel Performance Estimation

To provide an estimation of the parallel performance, a discrete event simulator is used. As shown in Fig. 28.8, the simulator takes as input (i) the time-annotated KPN traces, (ii) the MPSoC model (e.g., context switch and communication costs), and (iii) a mapping configuration. The latter can be provided by the user or generated automatically (see Sect. 28.5.4). The simulator *replays* the traces according to the mapping configuration and produces a Gantt-Chart and other execution statistics. This information can be used by the programmer to modify the application (or the architecture) and is also used by iterative mapping algorithms.

28.5.4 KPN Mapping

A detailed view of the mapping and scheduling phase is presented in Fig. 28.9. This phase starts with an initialization process in which, among others, a model of the execution of the application in form of a *trace graph* is created. After initialization, the mapping configuration is computed in an iterative way, as suggested by the figure. With every iteration, the mapper is allowed to use more resources in order to meet the constraints given by the user. Similar approaches have been described in [9, 13, 26, 30]. The outer iteration adds more processing elements (PEs), whereas the inner iteration makes more memory available, i.e., enlarges the FIFO buffers. After every iteration, the TRM is used to evaluate the performance of every mapping configuration. Further details are given in the following.

28.5.4.1 Initialization and Trace Graph Creation

During initialization, the starting amount of resources (processors and memories) for the iterative mapping process is determined. The starting set of processors and memories is defined by the platform constraints (see Sect. 28.5.1.1). The initial amount of memory is determined in turn by the minimal buffer sizes that lead to a deadlock-free execution.

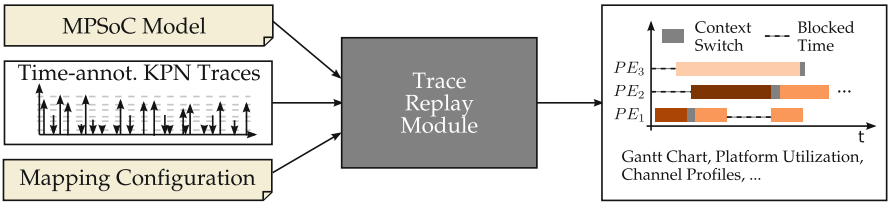


Fig. 28.8 Trace Replay Module (TRM)

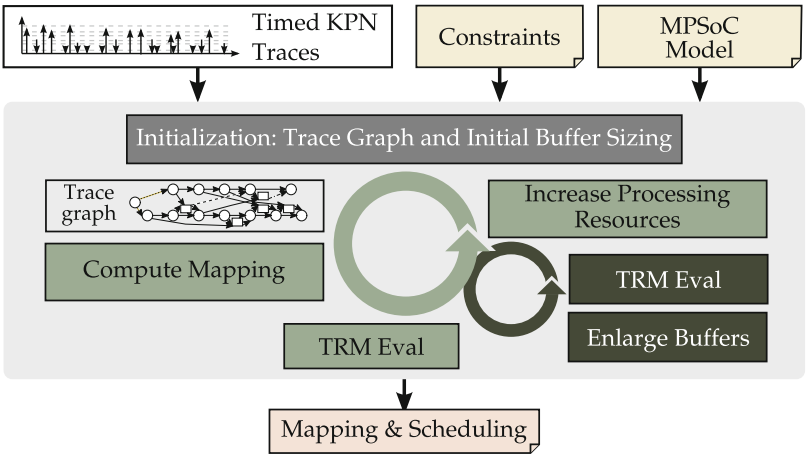


Fig. 28.9 KPN mapping flow

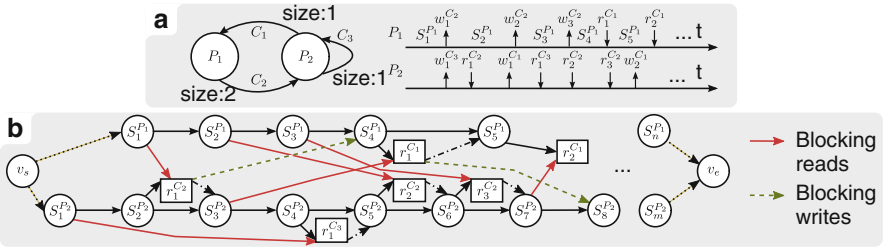


Fig. 28.10 Trace graph. (**a**) Sample KPN, annotated buffer sizes and sample trace with named segments. (**b**) Resulting trace graph

Plenty of authors have proposed buffer sizing strategies for KPN applications [5, 17, 28]. A common approach is to execute the application and selectively increase buffer sizes as the application reaches an artificial deadlock. We use the information collected in the traces to achieve a similar goal. To this end, a graph representation of the events observed during tracing is constructed. This so-called trace graph is then later used by mapping heuristics as discussed in Sect. 28.5.4.4.

In a trace graph nodes represent segments in the process traces and inter-process communication. The edges represent sequential execution of the segments and dependencies introduced by communication, i.e., a read can only succeed if the write has happened (blocking reads semantics) and a write can only succeed if there is enough space in the FIFO (blocking writes semantics), as Fig. 28.10 illustrates.

As can be seen from the example, the edges that represent blocking writes change with the buffer sizes. For example, the edge $(r_1^{C_2}, S_4^{P_1})$ states that the first read from channel C_2 unblocks the fourth segment of process P_1. This segment follows immediately after the third write to channel C_2, which would block if no token is

read, given the channel size of 2 in the example. If the buffer size for channel C_2 were 1, then the edge would be $(r_1^{C_2}, S_3^{P_1})$, i.e., it would unblock the second write to the channel. At the same time, a new edge $(r_2^{C_2}, S_4^{P_1})$ would appear. A wrongly dimensioned buffer would lead to deadlock, which appears as a cycle in the trace graph. For this reason, we compute the initial buffer sizes, by iteratively constructing the trace graph while ensuring absence of cycles.

28.5.4.2 Increasing the Amount of Processors

If a given platform subset is not enough to meet the constraints imposed by the user, the mapping process tries again with more resources or a new set of different ones. Note that for a platform with N processors, potentially of different type, this procedure requires the mapping process to be run for $2^N - 1$ subgraphs of the $MPSoC$ model. To avoid exponential complexity, hardware symmetries are exploited as described in [18] to ignore equivalent subgraphs. Complexity is further reduced by monotonically increasing the cardinality of the processor subset. That is, for every two subsequent subsets $\mathscr{PE}_i, \mathscr{PE}_{i+1} \in \wp(\mathscr{PE}), |\mathscr{PE}_i| \leq |\mathscr{PE}_{i+1}|$. To achieve this, we randomly select among all possible subsets at every iteration.

28.5.4.3 Enlarging Buffers

It is known that increasing buffer sizes may lead to a throughput improvement. To exploit this fact, the buffer sizes are increased in a non-uniform way for every platform subset. This is done by observing the TRM schedule, selecting the channel that was full for the longer time and increasing its size by the biggest burst access.

28.5.4.4 Mapping Computation

The MAPS framework includes several heuristics for mapping KPN applications, as described in [8, 9]. While several authors proposed evolutionary approaches to address the mapping problem as in ▸ Chap. 30, "DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips" of this book and in [29,31], here it was opted for simpler and faster heuristics based on the execution traces.

Given a $KPN = (\mathscr{P}, \mathscr{C})$ and an $MPSoC = (\mathscr{PE}, \mathscr{E})$, we want to determine a mapping of processes to processors $\mu_p : \mathscr{P} \rightarrow \mathscr{PE}$ and a mapping of channels to communication primitives $\mu_c : \mathscr{C} \rightarrow \mathscr{E}$, so that $\forall C = (P_i, P_j) \in \mathscr{C}$, with $\mu_c(C) = CP = (PE_k, PE_l) \in \mathscr{E}$, $\mu_p(P_i) = PE_k \wedge \mu_p(P_j) = PE_l$. The mapping heuristics strive at finding one such valid mapping, with the best performance given the available resources (see Fig. 28.9).

The group-based mapping algorithm (GBM) [10] is one prominent heuristic for jointly mapping processes and channels. This heuristic is based on the analysis of the critical path of the trace graph introduced in Sect. 28.5.4.1, as many other mapping algorithms for mapping Directed Acyclic Graphs (DAGs) [23]. The GBM algorithm works by iteratively reducing the group of resources (processors or communication primitives) to which an application element (process or channel) can be mapped. Given these groups, one can compute the ASAP and ALAP times of every node

in the trace graph, for example, using the average cost across all resource types in the group. This helps to identify all nodes that belong to the critical paths of the graph. Among these nodes, the GBM algorithm then determines the *bottleneck* application element. This element is the one that would contribute the most to the reduction of the critical path length if its group were reduced. Groups are reduced by removing slow resources, lowering the average cost of the nodes in the graph, thereby shortening the critical path. After reducing the group of the bottleneck element, new critical paths with new bottleneck elements may appear. This way, groups are iteratively reduced until only one resource type is left available (for critical elements) or no more improvements can be achieved. Having solved the assignment of heterogeneous resources, a simpler heuristic is used afterward to determine the final mapping (see [10] for details). This group-based strategy helps dealing with highly heterogeneous platforms, by reserving specialized resources (e.g., accelerators) for the most crucial application components.

28.6 Code Generation Flow

In this section, the compiler infrastructure of the CPN language (*cpn-cc*) is presented. The *cpn-cc* compiler is a source-to-source (CPN-to-C) compiler implemented based on Clang [24], which is the frontend of the LLVM compiler infrastructure. Figure 28.11 shows the CPN compiler infrastructure. The inputs are the CPN application itself (obtained with the *sequential programming flow* described in Sect. 28.4) and a configuration that describes how the CPN application is mapped to the target MPSoC (obtained with the *parallel programming flow* from Sect. 28.5). The *cpn-cc* compiler itself consists of three stages, namely, the frontend, generic and platform-specific transformations, and C code generation:

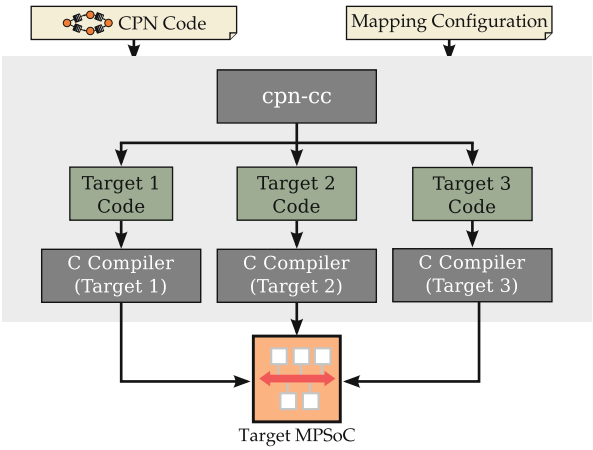


Fig. 28.11 CPN compiler infrastructure (*cpn-cc*)

- **Frontend**: The frontend starts with C preprocessing. The source code is then analyzed by the tokenizer where CPN keywords are added. The code is also analyzed by a parser that understands the new grammar rules for CPN syntax elements, such as process templates and channel accesses. The CPN-aware semantic analysis creates an *Abstract Syntax Tree (AST)* in the next stage. It will contain all CPN language elements that appear in the source code.
- **AST Transformations**: The AST transformations do the actual source-to-source translation. There are two categories for the AST transformations, *generic* and *platform-specific*. This is similar to the sequence of code optimizations in a classical compiler [4]. Generic transformations are platform independent, while platform-specific transformations replace the AST nodes of CPN constructs with C nodes that have platform-specific API calls according to the input mapping configuration (e.g., processes creation and FIFO communication primitives). The AST transformations result is a pure C AST that does not contain CPN elements.
- **C Code Generator**: The last step of the source-to-source translation is the AST printer of Clang that emits C source code from the transformed AST. Other additional auxiliary files such as makefiles are also generated to enable seamless compilation, using the native C compilers of the individual target cores.

The *cpn-cc* compiler provides a clean and powerful infrastructure for source code transformation based on ASTs. Code optimizations are possible thanks to the preservation of the complete semantic information. Data type checking and variable manipulation are enabled in the compilation, which is not soundly supported by other textual replacement approaches. This also results in good readability of the generated source code, as it is close to the original CPN specification.

28.7 Case Studies

This section describes case studies for the tool flows previously outlined in this chapter. The parallelization of Android applications and the mapping of embedded benchmarks from multiple application domains are presented as case studies for the sequential and parallel programming flow, respectively.

28.7.1 Parallelization of Android Software

The usefulness of a tool flow is directly related to the systems and software environments to which it can be applied. This section describes the extension to the sequential programming flow for parallelization of Android software [3]. Android is an interesting case study as it became the “cornerstone” of the major vendors in the mobile market [19]. One of the key aspects for this success is its Java-based programming model, which allows to develop portable applications across multiple hardware platforms. However, this portability comes at a price to be paid in terms of performance. To overcome this problem, the Android software

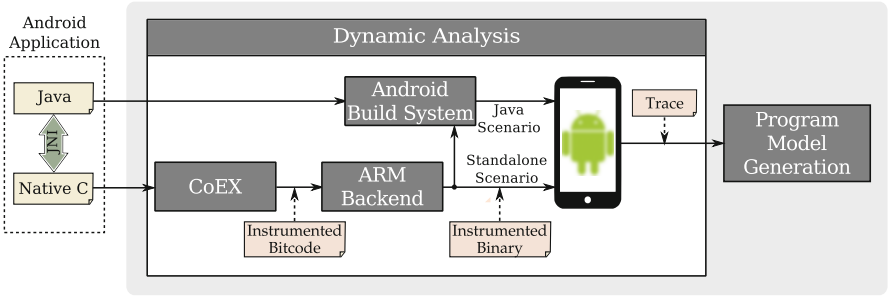


Fig. 28.12 Extension to the dynamic analysis to support android (The Android Robot created by Google is used under the terms of the Creative Commons 3.0 Attribution License)

Table 28.1 Characteristics of the benchmarks

Benchmark	LOC	Functions	Candidates	Parallelism
Beamformer	2 K	7	2	DLP
Edge detection	1 K	4	2	TLP, DLP
JPEG decoder	2 K	32	3	PLP
LTE	4 K	39	1	DLP
PNG decoder	27 K	170	8	PLP
Webp decoder	23 K	149	6	PLP

environment provides APIs (e.g., Java Native Interface JNI) that allow to develop computationally intensive portions of the applications (typically libraries) in native C or C++ code, which exploits specific characteristics of the target MPSoCs.

Figure 28.12 shows the extension to the sequential programming flow for Android-based devices. This is achieved by instrumenting the C portion of the Android applications. There are two possible scenarios: (i) the application is described as a combination of Java and C code (e.g., a library) using the JNI API, or (ii) the application is described purely in C. With this extension the usual Android development flow is not affected, since a set of scripts automate the whole process of instrumentation, execution, and trace retrieval from the device. To evaluate the performance of the sequential programming flow applied to Android, representative embedded applications were analyzed on the Nexus 7 tablet [27], which is based on a 1.5 GHz Quad-core Snapdragon MPSoC. For this evaluation the minimal threshold for parallelization candidates was set to 50% of the total execution time. Then the parallelism identified in the candidates was implemented using CPN taking into account the hints provided by the sequential programming flow.

Table 28.1 shows the characteristics of the benchmarks considered for the evaluation, in terms of the number of lines of code, number of functions, number of parallelization candidates identified, and the forms of discovered parallelism. The first observation from the table is that the number of parallelization candidates identified was on average less than four. This result shows that the workload of the applications is concentrated in few functions. This fact holds even for larger benchmarks, such as the WebP decoder, where only 4% of the functions were

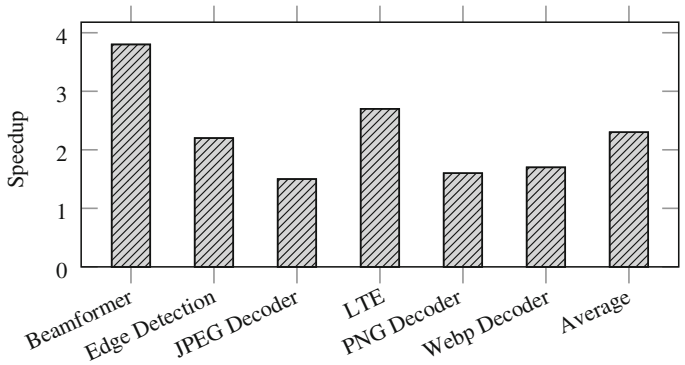


Fig. 28.13 Speedup results on the Nexus tablet

identified as parallelization candidates. This supports the idea that the sequential programming flow is scalable, as it focuses only on the parallelization candidates. Another interesting observation from Table 28.1 is that decoding benchmarks, such as JPEG, PNG, and WebP, profit from PLP. Such benchmarks have to process the compressed input bitstream, by performing a sequence of steps within a loop. The way this bitstream is processed prevents the exploitation of DLP. PLP extraction is fairly straightforward, as these steps or a combination of them can be implemented as pipeline stages.

Figure 28.13 shows the speedup results after comparing the time of the parallelized version of the applications, against the sequential version running on a single core. The average speedup gain among the evaluated benchmarks was 2.3×. The best utilization of the cores was achieved by the Beamformer benchmark with a speedup of 3.8×. This is because the workload of this benchmark is dominated by a loop that allows the extraction of DLP, thus its parallel representation scales very well on the four cores available. In particular, the result of the PNG benchmark is important in the context of Android, as PNG is the preferred image format in this framework. Therefore, the speedup gain on PNG impacts the entire Android framework. It is worth mentioning that after a manual inspection of the parallelization suggestions of the tool, no data dependency violations were identified, which could compromise the functional correctness of the parallelized benchmarks. Moreover, it was found through the case studies that the process of deriving the CPN representation is straightforward, when taking the provided hints into account.

28.7.2 Mapping of Multi-domain Embedded Benchmarks

In this section, the parallel programming flow is evaluated in terms of *performance* and *productivity* improvements [1]. The evaluation is conducted by exploring the CPN description mapping of the applications of an embedded benchmark set on

the C6678 multi-core DSP platform from Texas Instruments [7]. This MPSoC has eight DSP cores and offers multiple alternatives for communication, such as on-chip shared memory and a hardware controller for packet-based data movement.

The considered benchmarks for the evaluation belong to multiple application domains, such as audio, image and video processing, radar, and wireless communications. Table 28.2 shows a summary of their characteristics, such as the number of processes, FIFOs, and the forms of exploited parallelism. The mapping options column gives an idea of the mapping complexity. It considers both the characteristics of the applications, such as the number of processes and the number of FIFOs, the number of MPSoC processing elements (eight DSPs), and the types of available FIFOs (e.g., shared memory and multi-core navigator). The number of mapping options for each benchmark is computed with the expression: $(8)_{Processes} * (2)_{FIFOs}$.

First the performance is evaluated in terms of speedup and efficiency of every benchmark, by comparing the results of manual mapping against the automatic mapping provided by the parallel programming flow. Figure 28.14a summarizes the achieved speedups, taking as a baseline the sequential version of each benchmark on a single DSP core. Furthermore, it is possible to establish a relationship between the number of cores used in a given parallel implementation and the resulting speedup, by computing the achieved efficiency ($speedup/cores$). Figure 28.14b shows efficiency results both on the manual and automatic mapping scenarios. In general, the automatic mapping shows a better performance and efficiency results.

Table 28.2 Characteristics of benchmarks

Benchmark	Processes	FIFOs	Parallelism patterns	Mapping options
Audio filter	8	8	DLP, PLP	4.3×10^9
Sobel filter	8	10	TLP, DLP	1.7×10^{10}
JPEG	24	27	DLP, PLP	6.3×10^{29}
MJPEG	5	4	PLP	5.2×10^5
MIMO OFDM	18	23	DLP, PLP	1.5×10^{23}
STAP	16	18	DLP, PLP	7.4×10^{19}

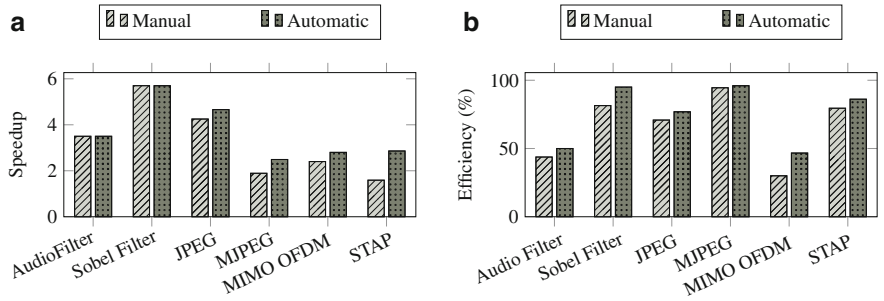


Fig. 28.14 Mapping results on the C6678 Keystone platform. (**a**) Speedup. (**b**) Efficiency

Table 28.3 Productivity results

Benchmark	Manual		Automatic		Productivity
	Trials	Time(h)	Trials	Time(s)	Gain
Audio filter	1	0.5	1	0.80	2250
Sobel filter	3	1.5	1	0.57	3440
JPEG	6	3	1	2.00	5400
MJPEG	2	1	1	1.46	4932
MIMO OFDM	15	7.5	1	2.06	13,107
STAP	10	5	1	3.20	5625

This is especially true in complex KPNs, such as the JPEG, MIMO OFDM, and STAP, where manual mapping is not able to achieve the best utilization of the available DSP cores.

To evaluate the productivity gains provided by the parallel programming flow, the time consumed to perform manual and automatic mapping is compared. While automatic mapping usually requires just one execution of the tool flow, manual mapping requires not only a priori knowledge of the benchmarks but also a considerable effort as multiple trials are needed to find a proper result. Each trial of manual mapping is an expensive task that involves defining a configuration, generating code for the target device, executing it on the platform, and evaluating the resultant performance. The duration of each manual mapping iteration is estimated as approximately 30 min. Therefore, the time required to obtain a mapping solution based on this approach could be in the order of hours. The achieved productivity gains can be computed as the ratio of the time it takes to manually find a suitable mapping, over the time the parallel programming flow consumes to achieve one automatically.

Table 28.3 shows the productivity results. The highest productivity is achieved with complex benchmarks, such as JPEG, MIMO OFDM, and STAP. The benefit of using the tool flow is clear, since a mapping configuration is generated in a matter of seconds with one single execution of the tool flow, while manual mapping requires several trials that could take hours to achieve acceptable performance results.

28.8 Silexica: The Industrial Perspective

Programming multi-core computing systems has been known as a challenging problem for decades. A great amount of efforts have been placed to address this problem by the academia. Nevertheless, the industry is yet to benefit from the research achievements. Converting academic results into industrial practice requires significant amount of resources. Martijn De Lange (ACE) said in the *Software*

Tools for Next Generation Computing organized by the European Commission in June 2014 that a full consistent heterogeneous parallel programming environment is estimated to require 80M€ initial development budget.

In this section, we take a look at the *gap* between the state-of-the-art research results and industrial requirements for multi-core programming tools from the perspective of Silexica, a start-up company providing multi-core software tools. By sharing our early experience of commercializing the multi-core compiler research results from MAPS, we explain how to bridge the gap not only *technically* but also *operationally*. This short report serves as an industrial perspective in preparing and planning future research roadmaps in multi-core computing research.

- The fundamental problem why current programming languages fall short for multi-core computing systems is the lack of parallelism support. A natural solution, which many academic approaches pursued, is to design new languages that explicitly support parallelism and the necessary run-time information. While the introduction of new information is absolutely necessary, the new language approaches may hinder industry adoption as there is too much legacy software that has been accumulated from the past decades.

  Though a new language approach could be cleaner and more elegant in expressing parallelism semantics, the pragmatic first step is to support parallelism by extending existing ones. CPN, a lightweight C-based dialect, was developed to introduce model-based parallelism (See Sect. 28.2). Silexica also provides a partitioning tool, which helps end users to gradually convert and parallelize legacy C sequential code to the new dialect. This closeness to C and gradual software migration are favored by industry partners and certainly helped opening the door to further opportunities to improve the tools.
- The introduction of parallelism support depends upon a use-model change for end users. It means that often the existing software (or a software building environment) has to be modified either to enable compiler analysis for more parallelism or deploy parallelized results. Academic tools tend to overlook this problem, leading to a large overhead incurred by this use-model change. The first user experience by early industrial adopters suffers to some extent due to this.

  To cope with the use-model change by introducing new compilation tools, one possible solution is a getting-started kit for adopters to gradually convert their code base to use new tools. This could be, e.g., a prepared project structure, which migrates current user software into a new environment. Or in more complicated scenarios, current user software has to be divided into parts where a new approach could apply to each part as a pilot project. Our experience also showed that, as the compilation process for multi-core platforms is complex and long, it is important to provide users an intuitive (preferably graphical) user interface with comprehensive debugging information to improve the user experience.
- Academic works in compiler research are usually demonstrated by using some benchmarks on defined architectures. Unlike the relatively small number of

architecture types in the uni-processor time, the complexity and diversity nowadays in multi-core platforms is unprecedented. Most of those platforms are either expensive to acquire or company proprietary, which makes them hardly accessible by universities. What makes it even worse is that there are no established benchmark suites for multi-core compilers which allow horizontal comparison.

Though multi-core benchmark suites (e.g., MultiBench from EEMBC) have started to get attention recently, it is hard to expect that those benchmarks would be ported to a fairly large number of target platforms due to the amount of work required. In general, it is difficult, if not impossible, to benchmark heterogeneous multi-core platforms using simple metrics through common benchmarks. It is quite difficult for industrial companies to evaluate academic results. To overcome the problem of lacking reference comparison for academia to gain credit from industry, it would be beneficial to be able to use industry benchmarks, possibly through collaboration or professional associations early in the research cycle.

28.9 Conclusion

This chapter presented the MPSoC Application Programming Studio (MAPS), which is a framework to close the software productivity gap for heterogeneous MPSoCs. MAPS is composed of multiple tool-flows for parallelization, mapping/scheduling and code generation. In this chapter, two case studies were also presented to show the applicability of MAPS in commercial environments. This chapter closed by discussing early experiences of transferring the MAPS technology into the industrial practice through Silexica GmbH, a start-up company that provides programming solutions for multicore embedded systems.

References

1. Aguilar M, Jimenez R, Leupers R, Ascheid G (2014) Improving performance and productivity for software development on TI multicore DSP platforms. In: 6th European embedded design in education and research conference (EDERC), 2014, pp 31–35
2. Aguilar MA, Eusse JF, Leupers R, Ascheid G, Odendahl M (2015) Extraction of kahn process networks from while loops in embedded software. In: 12th IEEE international conference on embedded software and systems (ICESS)
3. Aguilar MA, Eusse JF, Ray P, Leupers R, Ascheid G, Sheng W, Sharma P (2015) Parallelism extraction in embedded software for Android devices. In: Proceedings of the XV international conference on embedded computer systems: architectures, modeling and simulation, SAMOS XV
4. Aho AV, Lam MS, Sethi R, Ullman JD (2006) Compilers: principles, techniques, and tools, 2nd edn. Prentice Hall, Boston
5. Basten T, Hoogerbrugge J (2001) Efficient execution of process networks. In: Chalmers A, Mirmehdi M, Muller H (eds) Communicating process architectures – 2001. IOS Press, Amsterdam, pp 1–14

6. Brunet SC (2015) Analysis and optimization of dynamic dataflow programs. Ph.D. thesis, Ecole Polytechnique Federale de Lausanne (EPFL)
7. C6678: Multicore fixed and floating-point digital signal processor. http://www.ti.com/product/TMS320C6678/technicaldocuments
8. Castrillon J, Leupers R (2014) Programming heterogeneous MPSoCs: tool flows to close the software productivity gap. Springer, Cham
9. Castrillon J, Leupers R, Ascheid G (2013) MAPS: mapping concurrent dataflow applications to heterogeneous MPSoCs. IEEE Trans Ind Inf 9(1):527–545
10. Castrillon J, Tretter A, Leupers R, Ascheid G (2012) Communication-aware mapping of KPN applications onto heterogeneous MPSoCs. In: Proceedings of the 49th annual design automation conference, DAC'12. ACM, New York, pp 1266–1271
11. Castrillon J, Velasquez R, Stulova A, Sheng W, Ceng J, Leupers R, Ascheid G, Meyr H (2010) Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms. In: Proceedings of the conference on design, automation and test in Europe, DATE'10. European design and automation association, pp 753–758
12. Ceng J, Castrillon J, Sheng W, Scharwächter H, Leupers R, Ascheid G, Meyr H, Isshiki T, Kunieda H (2008) MAPS: an integrated framework for MPSoC application parallelization. In: Proceedings of the 45th annual design automation conference. ACM, pp 754–759
13. Cheung E, Hsieh H, Balarin F (2007) Automatic Buffer Sizing for Rate-constrained KPN applications on multiprocessor System-on-Chip. In: Proceedings of the 2007 IEEE international high level design validation and test workshop. IEEE, pp 37–44
14. Das A, Singh AK, Kumar A (2015) Execution trace–driven energy-reliability optimization for multimedia MPSoCs. ACM Trans Reconfigurable Technol Syst 8(3):18:1–18:19
15. Eusse JF, Williams C, Leupers R (2015) CoEx: a novel profiling-based algorithm/architecture co-exploration for ASIP design. ACM Trans Reconfigurable Technol Syst 8(3):17:1–17:16
16. Eusse J, Williams C, Murillo L, Leupers R, Ascheid G (2014) Pre-architectural performance estimation for ASIP design based on abstract processor models. In: International conference on embedded computer systems: architectures, modeling, and simulation (SAMOS XIV), 2014, pp 133–140
17. Geilen M, Basten T (2003) Requirements on the execution of kahn process networks. In: Proceedings of the 12th European symposium on programming, ESOP 2003. Springer, pp 319–334
18. Goens A, Castrillon J (2015) Analysis of process traces for mapping dynamic kpn applications to mpsocs. In: Proceedings of the IFIP international embedded systems symposium (IESS), Foz do Iguaçu
19. International Data Corporation (IDC) (2015) IDC: smartphone OS market share, 2015 Q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp
20. Johnson R, Pingali K (1993) Dependence-based program analysis. In: Proceedings of the ACM SIGPLAN 1993 conference on programming language design and implementation, PLDI'93. ACM, New York, pp 78–89. doi:10.1145/155090.155098
21. Kahn G (1974) The semantics of a simple language for parallel programming. In: IFIP congress, pp 471–475
22. Kennedy K, Allen JR (2002) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco
23. Kwok YK, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput Surv 31(4):406–471. doi:10.1145/344588.344618
24. Lattner C (2008) LLVM and clang: next generation compiler technology. In: The BSD conference, Ottawa
25. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
26. Moreira O, Valente F, Bekooij M (2007) Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In: EMSOFT'07: Proceedings of the 7th ACM & IEEE international conference on embedded software. ACM, pp 57–66
27. Nexus 7 (2013) http://www.asus.com/Tablets_Mobile/Nexus_7_2013/

28. Parks TM (1995) Bounded scheduling of process networks. Ph.D. thesis, EECS Department, University of California, Berkeley
29. Pimentel A, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112
30. Stuijk S, Basten T, Geilen MCW, Corporaal H (2007) Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In: DAC'07: Proceedings of the 44th annual design automation conference. ACM, New York, pp 777–782
31. Thiele L, Bacivarov I, Haid W, Huang K (2007) Mapping applications to tiled multiprocessor embedded systems. In: International conference on application of concurrency to system design, pp 29–40. doi:10.1109/ACSD.2007.53
32. Thies W, Chandrasekhar V, Amarasinghe S (2007) A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In: Proceedings of the 40th annual IEEE/ACM international symposium on microarchitecture, MICRO 40. IEEE, pp 356–369
33. Thies W, Karczmarek M, Amarasinghe S (2002) StreamIt: a language for streaming applications. In: International conference on compiler construction, Grenoble
34. Tournavitis G (2011) Profile-driven parallelization of sequential programs. Ph.D. thesis, University of Edinburgh

HOPES: Programming Platform Approach for Embedded Systems Design

29

Soonhoi Ha and Hanwoong Jung

Abstract

Hope Of Parallel Embedded Software (HOPES) is a design environment for embedded systems supporting all design steps from behavior specification to code synthesis, including static performance estimation, design space exploration, and HW/SW cosimulation. Distinguished from other design environments, it introduces a novel concept of "programming platform" called Common Intermediate Code (CIC), which can be understood as a generic execution model of heterogeneous multi-processor architecture. In the CIC model, each application is specified by a multi-mode Synchronous Data Flow (SDF) graph, called MTM-SDF. Each mode of operation is specified by an SDF graph and mode transition is expressed by an Finite-State Machine (FSM) model. It enables a designer to estimate the performance and resource demand by constructing static schedules of the application with varying number of allocated processing elements at each mode. At the top level, a process network model is used to express concurrent execution of multiple applications. A special process, called control task, is introduced to specify the system-level dynamism through an FSM model inside. With a given CIC model and a set of candidate target architectures, HOPES performs design space exploration to choose the best HW/SW platform, assuming that a hybrid mapping policy is used to map the applications to the processing elements. HOPES synthesizes the target code automatically from the CIC model with the mapping information. The overall design flow is verified by the design of two real-life examples.

S. Ha (✉)
Department of Computer Science and Engineering, Seoul National University, Gwanak-gu, Seoul, Korea
e-mail: sha@snu.ac.kr

H. Jung
Seoul National University, Gwanak-gu, Seoul, Korea
e-mail: jhw7884@gmail.com

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_1

Acronyms

API	Application Programming Interface
BDF	Boolean Data Flow
CIC	Common Intermediate Code
FSM	Finite-State Machine
GA	Genetic Algorithm
GUI	Graphical User Interface
KPN	Kahn Process Network
MTM	Mode Transition Machine
NoC	Network-on-Chip
OS	Operating System
PIM	Platform Independent Model
PSDF	Parameterized Synchronous Data Flow
QoS	Quality of Service
SADF	Scenario-Aware Data Flow
SDF	Synchronous Data Flow
SMP	Symmetric Multi-Processing
SysteMoC	SystemC Models of Computation
WCRT	Worst-Case Response Time

Contents

29.1 Introduction

HOPES is under development as a generic design environment to support a wide range of embedded system architectures from system on a chip (SoC) to networked embedded systems. Starting from a target-independent behavior specification and a given set of candidate hardware architectures and available processing elements, we can explore the design space to find an optimal system configuration and mapping of applications, and synthesize the software and hardware components in a unified framework. The abstract target architecture assumed in HOPES consists of heterogeneous processing elements that are connected through a network. HOPES

was originally introduced as a parallel programming environment for nontrivial heterogeneous multiprocessors with various design constraints on hardware cost, power, and real-time performance [15]. However, HW/SW codesign can be naturally supported by HOPES, since a hardware IP can be regarded as a processing element.

As the system complexity incessantly grows with more processing elements integrated, design reuse of hardware platforms and IPs becomes the de facto practice to mitigate the difficulty of hardware validation. Then the HW/SW codesign methodology is transformed to an embedded SW development methodology for a given hardware platform. Since the proportion of software components keeps increasing, HOPES puts more emphasis on the implementation of software components unlike our previous HW/SW codesign environment, PeaCE (Ptolemy extension as a Codesign Environment) [7]. While PeaCE focuses on the codesign of hardware and software modules that includes HW/SW partitioning, HW/SW cosynthesis, and HW/SW cosimulation, it takes little account of multi-processor architecture that heavily affects the parallel execution of software.

A systematic design methodology can be understood as a sequence of steps that refine a higher level of abstraction to a lower level from initial specification to final implementation, which is summarized as the following phrase: "design is to represent". Since refinement keeps the properties of the higher abstraction, how to specify the behavior is a key factor to distinguish various HW/SW codesign methods. Actor models that specify an application as a set of concurrent actors are widely adopted in the HW/SW codesign methodology since they express the potential parallelism of an application explicitly and parallelizing an application can be simply realized by mapping actors to the processing elements. Actors are connected to each other through channels that represent the flow of data samples between actors. Among many actor models, Synchronous Data Flow (SDF) is chosen as the baseline actor model of HOPES since it is a formal model that enables us to evaluate each design decision through static analysis. General introduction to data-flow models can be found in ▸ Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign".

In the SDF model [17], an application is specified with a data-flow graph where a node represents a function module, or a task, and an arc is a FIFO queue that delivers data samples from an output port of the source node to an input port of the destination node. An input (or an output) port is associated with an integer number that indicates how many samples to consume (or to produce) per task execution; the number is called the sample rate of the port. Figure 29.1a shows a simple SDF graph representation of an application. A node becomes runnable only when all input arcs have no fewer samples queued than the specified sample rate. And the sample rates are fixed at run time in the SDF model. Then we can determine the mapping and scheduling of the SDF graph, which is to determine where and in what order to execute the tasks on a given target architecture, at compile time. For each arc, we can determine the relative execution rates between the source task and the destination task, comparing the output sample rate of the source port and the input sample rate of the destination port. For instance, the execution rate of task C should be twice

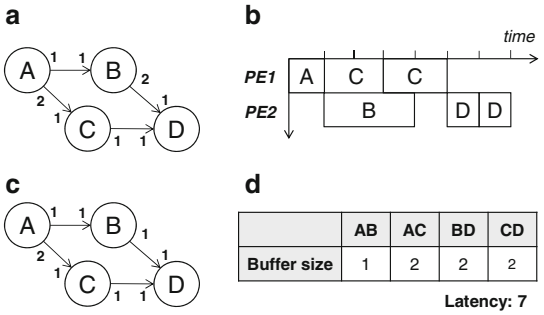


Fig. 29.1 (**a**) An example SDF graph with annotated sample rates on the arcs, (**b**) a mapping and scheduling result of the SDF graph onto two processing elements, (**c**) an example SDF graph that has a buffer overflow error, and (**d**) the buffer requirement of each arc and the estimated latency for the static scheduling result of (**b**)

higher than that of task *A* in Fig. 29.1a, in order to make the number of samples produced from the source task the same as the number of samples consumed by the destination task. An *iteration* of an SDF graph is defined by the set of task executions that satisfy the relative execution rates of tasks with minimum number of executions.

An SDF graph is said to be *consistent* if the relative execution rates of tasks are satisfied for all arcs. In case there is any possibility of buffer overflow on any FIFO arc or deadlock, we cannot find a valid schedule of an SDF graph. Figure 29.1b shows an example of a static scheduling result on a target architecture with two processing elements. For a consistent SDF graph, we can repeat the schedule iteratively without buffer overflow. Figure 29.1c shows an SDF graph example that has a buffer overflow error on arc *AC* by giving a wrong sample rate at the output port of node *B*. Such static analyzability is a very desirable feature for embedded system design since it enables us to detect a class of design errors by static analysis [17]. Moreover, once the mapping and scheduling decision is made, we can determine the buffer requirements for all arcs and estimate the real-time performance of the application. We can easily check whether the design constraints on the hardware requirement and real-time performance can be satisfied or not. For instance, the buffer requirement and the estimated latency associated with the static scheduling result of Fig. 29.1b is summarized in Fig. 29.1d.

While the SDF model has the aforementioned benefits from its static analyzability, it has severe limitations to be used as a general model for behavior specification. First, it is not possible to specify the dynamic behavior of an application since the sample rate of a port may not change dynamically. Second, it does not allow the use of shared memory for inter-node communication since the access order to the shared memory may change depending on the execution order of nodes. So the synthesized code may require much larger memory than a manually written code that usually uses shared variables for communication between function modules. To overcome those limitations, we have proposed several extensions to the SDF model in HOPES.

We use the Finite-State Machine (FSM) model in combination with the SDF model to express the dynamic behavior of an application [10]. Furthermore, a special actor, called library actor [20], is introduced to handle shared resources efficiently without side effects.

Following the heritage of heterogeneous modeling of Ptolemy [3] and PeaCE [7], HOPES uses a process network model at the top level to express concurrent execution of multiple applications. An application is modeled as a single process at the top level while the internal behavior of an application is specified by the extended SDF model. The system-level dynamic behavior is specified by a control task whose behavior is specified by the FSM model in the top-level task graph. The overall specification model of HOPES is called the Common Intermediate Code (CIC) model, which will be explained in the next section in detail.

There is a clear distinction between HOPES and the other model-based design environments. As the name implies, the CIC model is not defined as a front-end specification model, but an intermediate specification model, meaning that HOPES design environment can accommodate various front-end specification models as long as the front-end specification model can be translated into the CIC model. In fact, the CIC model can be understood as an *execution* model of tasks at the Operating System (OS) level. At the OS level, the system behavior is represented as a set of tasks no matter what the front-end specification model is. Communication and synchronization between tasks and scheduling of tasks are heavily dependent upon the underlying software platform and hardware platform. In HOPES, we propose to define the execution model of tasks at the OS level and enforce the system to keep the semantics of the execution model. Then, the CIC model will be able to run on any hardware and software platform since the execution model is defined as platform-independent. So, we introduce a new notion of *programming platform* that hides the underlying software and hardware platform from the application programmer. As a program based on the von Neumann execution model can be run on any von Neumann processor, any program based on the CIC execution model can be run on any target architecture that keeps the CIC execution model, we envision. Since the CIC model is based on formal models of computation, we can enjoy the benefits of static analyzability of those models to reduce the design time and cost.

Even though the same SDF model is used for behavior specification in HOPES and PeaCE, the granularity of a node is quite different. In HOPES, an SDF node is a unit of mapping and scheduling at the OS level. It implies that the node granularity should be as large as a thread to make the thread switching overhead insignificant. On the other hand, PeaCE assumes mixed granularity of SDF nodes in the initial specification of an application and clusters them to define a thread or a task at the code synthesis step. In our previous work [15], an SDF graph specification of PeaCE has been translated to the CIC model by clustering the nodes to increase the granularity while keeping the potential parallelism as much as possible. It is possible to specify the system behavior with the CIC model manually, regarding the CIC model as the front-end specification. In this case, it is the responsibility of the programmer to define the granularity of the node to trade-off the parallelism and the scheduling overhead.

In the conventional model-driven software development approaches, the system behavior is specified by a Platform Independent Model (PIM) that is translated into a platform-specific model (PSM) manually for a given hardware platform. Then the target code is generated from the translated PSM. Even though the CIC model is a platform-independent model, there is no need to translate it to a PSM since the CIC model can be executed in any hardware platform that supports the proposed execution model. The HOPES framework is distinguished from other model-based design frameworks that use a specific model of computation for behavior specification, which include Daedalus [19], DAL [21], CompSOC [6], and Koski [12]. On the other hand, as mentioned above, HOPES does not assume any specific model for behavior specification as long as it can be translated into the CIC model. Even though the CIC model is based on three different models of computation, its model composition rule is different from that of Ptolemy [3] which allows hierarchical composition of models without limitation on the depth of hierarchy and on the kinds of models.

Figure 29.2 shows the overall design flow in HOPES. The input information consists of the front-end specification of system behavior and the set of candidate platforms and hardware components. As explained above, the CIC model is generated manually or by an automatic translator from a different front-end specification of system behavior. We perform static analysis at the CIC level to detect the buffer overflow and deadlock errors for SDF specifications and to analyze the timing requirements that will be expressed in the proposed FSM model. The next step is to explore the architectural design space by selecting the hardware platform and processing elements and mapping the applications to the target architecture. Note that the profiling information of tasks for all kinds of candidate processing elements is assumed to be given. The DSE step produces a handful of selected target architectures and associated mapping results of applications. Note that if a target

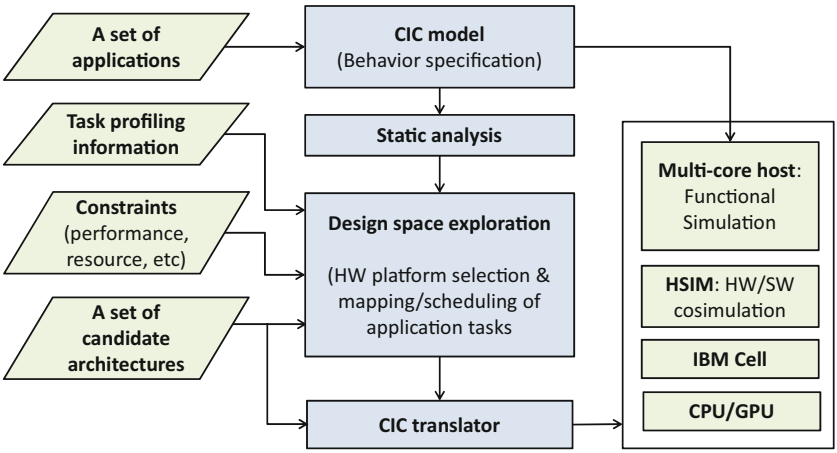


Fig. 29.2 Design flow of HOPES

architecture is given as an input, it just determines the mapping of applications to the target architecture.

For each candidate solution, the CIC translator generates a target C code for each processor. We need to develop a separate CIC translator for each target architecture as we need a different compiler for a different von Neumann processor. A multi-core host processor is a base target platform for functional simulation. The CIC translator generates a multi-threaded C code for functional simulation. Another target platform that HOPES supports is the parallel simulator, called HSIM [26], that has been developed to simulate the target architecture without real hardware platform. A handful of selected architectures will be evaluated more accurately by HSIM simulation. Other target platforms that HOPES supports will be explained later.

The rest of this chapter is organized as follows. The CIC model will be explained in the next section, which will be followed by Sect. 29.3 that explains the mapping and scheduling techniques of the CIC model. Section 29.4 explains the CIC translator. Preliminary experimental results will be discussed in Sect. 29.5. The current status of HOPES development is presented with concluding remarks in Sect. 29.6.

29.2 Common Intermediate Code (CIC) Model

As explained above, the proposed CIC model adopts a hierarchical composition of different models of computation to express the system behavior at two different levels. At the top level, the CIC model expresses the system behavior with a process network. If an application can be specified by the extended SDF graph, the application is encapsulated as a super node that contains an extended SDF graph at the bottom level. Note that the top-level process network and the extended SDF model themselves can be specified in a hierarchical fashion.

The top-level process network consists of CIC tasks and channels as depicted in Fig. 29.3. There are two types of CIC tasks depending on the triggering condition of tasks: *time-driven* and *data-driven*. A time-driven task is triggered by a pre-defined period that is given as a parameter. So it consumes the most recent sample from the input buffer channel. The input channels of a time-driven task are single-entry buffers that store the most recent data samples. An I/O task that interfaces with

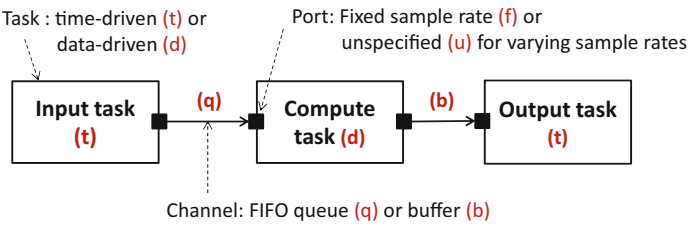


Fig. 29.3 CIC task graph

```
TASK_INIT{ /* task initialization code */ };

TASK_GO {
   /* generic API for data read from an input port */
   MQ_RECEIVE(port_name, data, size);
   ...
   /* generic API for system service request */
   SYS_REQ(command, argument_list);
   ...
   /* generic API for data write to an output port */
   MQ_SEND(port_name, data, size);
}

TASK_WRAPUP { /* task wrapup code */ };
```

Fig. 29.4 CIC task code template

the outside is usually designated as a time-driven task. On the other hand, a data-driven task is triggered by the arrival of data samples on the input ports. The input channels of data-driven tasks are assumed to be FIFO queues. A data-driven task basically follows the semantics of the Kahn Process Network (KPN) model that performs blocking read and non-blocking write access to the channels.

As shown in Fig. 29.4, the code template of a CIC task consists of three sections, enclosed by three keywords, TASK_INIT, TASK_GO, and TASK_WRAPUP. As the name implies, the TASK_INIT section is executed when the task is initialized and the TASK_WRAPUP section is executed just before it is terminated. The TASK_GO function is the main body that will repeat until the task is terminated. A CIC task accesses a channel with target-independent generic APIs, MQ_SEND and MQ_RECEIVE. The MQ_RECEIVE API performs blocking read operation to the associated input port while the MQ_SEND API performs non-blocking write operation to the associated output port. Since the CIC model is defined at the OS level, the CIC task assumes that there is a supervisor that schedules the CIC tasks and provides supervisory services to the CIC tasks. Thus, we define another generic API, SYS_REQ, that requests a service to the supervisor. The first argument of the SYS_REQ API defines the service command whose list will be shown later. In principle, a CIC task does not use platform-specific APIs for portability. The generic APIs will be translated into target-specific APIs at the code generation step. We may define a CIC task that uses platform-specific APIs for efficient implementation at the expense of portability.

The number of data samples consumed or produced per execution of a task can be specified explicitly for each input or output port. The sample rate is specified, if it is fixed and not changing at run time. Otherwise, the sample rate is assumed to be varying at run time. If the input sample rates of all input ports are specified, the data-driven task becomes an SDF task that follows the execution semantics of the SDF model. If all tasks in a CIC subgraph are SDF tasks, the CIC subgraph becomes an SDF subgraph. Since the SDF model has many merits from static analyzability, it is highly recommended to identify SDF subgraphs as much as possible at the top level until no more SDF subgraph can be identified. And each subgraph is replaced

by a super node at the top level to make it a two-level hierarchical graph. To alleviate the difficulty of identifying the SDF subgraph automatically, it is recommended to specify an application with the extended SDF model manually inside a super node.

29.2.1 Extended SDF Model for Application Specification

In this subsection, we explain a couple of extensions that are made to overcome the limitations of the SDF model while preserving the benefits of static analyzability. The first extension is to use the FSM model to express the dynamic behavior of the application. The second is to introduce a special actor, called library actor, to allow tasks to share HW or SW resources.

29.2.1.1 Dynamic Behavior Specification

There exist several approaches that have been proposed to increase the expression capability of the SDF model to support intra-application dynamism. One approach is to extend the SDF model itself. Dynamic Data Flow (DDF) and Boolean Data Flow (BDF) are two examples of this approach where they introduce special kinds of nodes that may have varying sample rates [2]. Since BDF was proven to be Turing equivalent and DDF is a super set of BDF, their expression capability is maximal in theory. But they compromise some benefits of static analysis and efficient implementation.

Another approach is to express the dynamism of an application as a set of *mode*s that the application takes and each mode is specified by an SDF graph. This approach assumes that the number of possible dynamic behaviors, or modes, is finite. Then the dynamic behavior can be expressed as dynamic mode change. There are several ways to specify mode change. In Parameterized Synchronous Data Flow (PSDF), the dynamic behavior of a task is modeled by parameters, and the mode change is realized by changing the parameters at run time before starting an iteration of a schedule [1]. An application is specified by a tuple of graphs, init graph and body graph, where the body graph specifies the application behavior and the init graph sets the parameter values to change the mode before a new iteration starts.

The other way is to combine the SDF model with another computation model, usually FSM to express the mode change. In the *-chart approach [5], each state of a finite state machine contains an SDF graph inside to make a hierarchical composition of SDF and FSM models. The state change in the FSM can be understood as the mode change of the SDF model. In the SystemC Models of Computation (SysteMoC) approach [9], a task is associated with an FSM that determines the execution behavior of the task. FSM-based SADF, shortly FSM-SADF, is a restricted form of Scenario-Aware Data Flow (SADF) that specifies each mode of operation, called scenario, with an SDF graph [22, 23]. An SDF task may have multiple versions of definition depending on the mode of operation while a special control actor, called detector, that has an FSM inside sends the control information to normal SDF tasks to change the mode of operation.

HOPES uses a similar approach as FSM-SADF; an SDF task may have multiple behaviors and a tabular specification of an FSM, called Mode Transition Machine (MTM), describes the mode transition rules for the SDF graph. An MTM is defined as a tuple {*Modes, Variables, Transitions*} where *Modes* and *Variables* represent a set of modes and a set of mode variables respectively, and *Transitions* is a set of transitions that consists of the current mode, a Boolean function of conditions, and the next mode. A Boolean function of transition condition is defined by a simple comparison operation between a mode variable and a value. An MTM-SDF specification of an H.264 decoder is shown in Fig. 29.5. The H.264 decoder has two modes of operation: I-frame and P-frame. In the I-frame mode, the sample rate of each port in red boxes becomes zero while the sample rate of each port in blue boxes becomes zero in the P-frame mode. The MTM is quite simple since it needs to distinguish two modes of operation by a single mode variable. Remind that the granularity of a CIC task is large and the dynamic behavior inside a task is not visible at the CIC level. Thus, an MTM is not complex in general for stream-based applications.

Mode transition is enabled by setting the mode variable so as to satisfy the transition condition. But actual mode transition occurs only at the iteration boundary of the SDF schedule. Since an SDF graph has a well-defined notion of iteration and each task knows how many times it should be executed in each iteration, mode transition can be performed autonomously by individual tasks without global timing synchronization. A mode variable can be set by the hidden supervisor, which will be discussed in the next subsection. Or a designated task may set the mode variable. A stream-based application usually starts with parsing a header information that determines the mode of operation, followed by processing a stream of data. In this case, the SDF task that parses the header information is designated as a special task that may change the mode variable. To satisfy the restriction that the mode transition occurs at the iteration boundary, the designated task should be the first task in the SDF schedule. In the H.264 decoder of Fig. 29.5, *RealFileH* is designated as the special task that determines the mode of operation.

The internal behavior of an SDF task should be defined manually depending on the mode of operation. The code skeleton of an MTM-SDF task is shown in Fig. 29.6; a task first checks the current mode of its MTM before starting the next iteration. If it is designated as a special task, it may change the mode variable. Based on the mode of operation, the sample rates of SDF graph can be changed. For instance, the sample rates for the input and output arcs of *IntraPredY/U/V* tasks become all zero for the P-frame mode and the sample rates for the output arcs of *InterPredY/U/V* tasks become zero for the I-frame mode of operation. Note that the feedback input arcs of *InterPredY/U/V* tasks do not change the sample rates since they need to store the previous frame fed back from the *Deblock* task even in the I-frame mode.

29.2.1.2 Library Task

In addition to dynamic behavior specification, another extension is made to the SDF graph by introducing a special task, called library task, to allow the use of

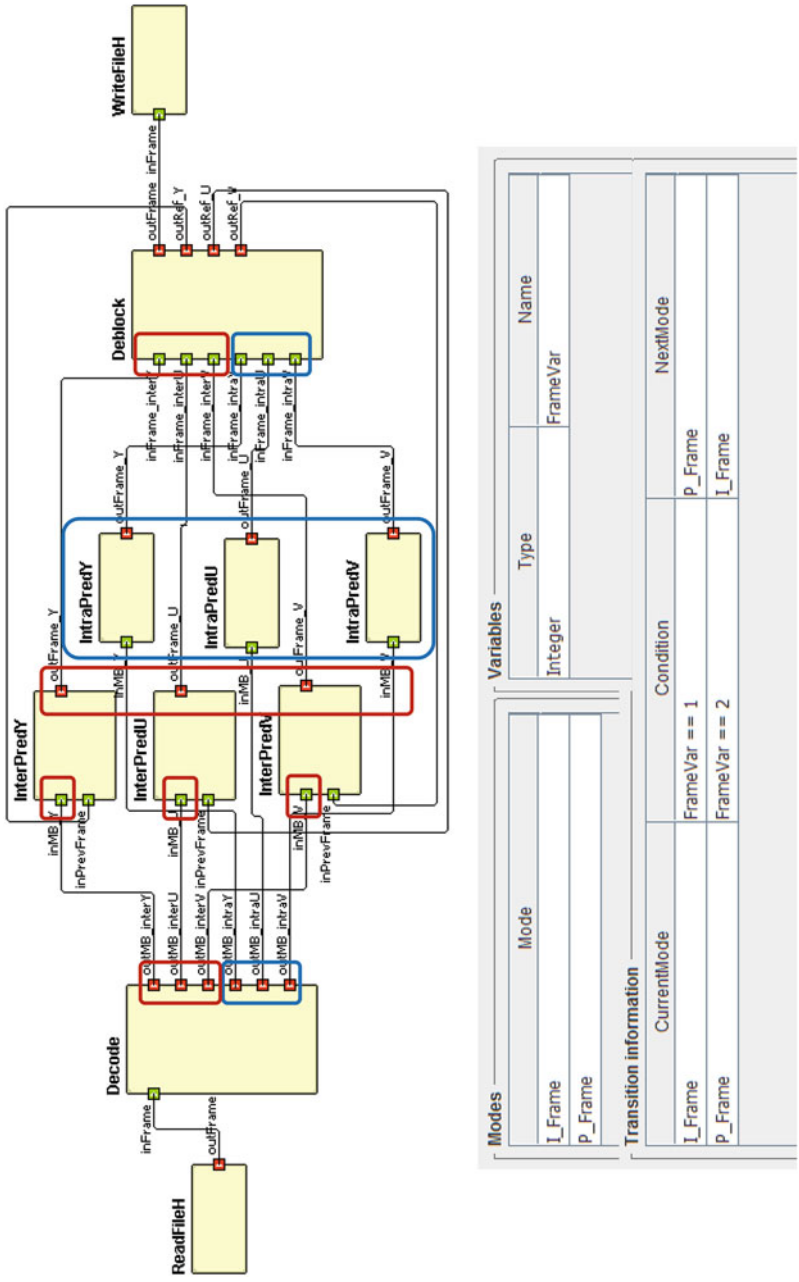


Fig. 29.5 An MTM-SDF specification of H.264 decoder: captured screen from the HOPES environment

```
TASK_GO{
   Mode = SYS_REQ(GET_CURRENT_MODE_NAME); // get a current mode
   if Mode == "S1":                       // code for mode S1
     MQ_RECEIVE(port_in, data, size);
     …
     MQ_SEND(port_out, data, size);
   else if Mode == "S2":                  // code for mode S2
     …
   if specific conditions:          // set a variable in an MTM
     SYS_REQ(SET_MTM_PARAM_INT, task_name, var_name, value);
}
```

Fig. 29.6 Code skeleton of an MTM-SDF task

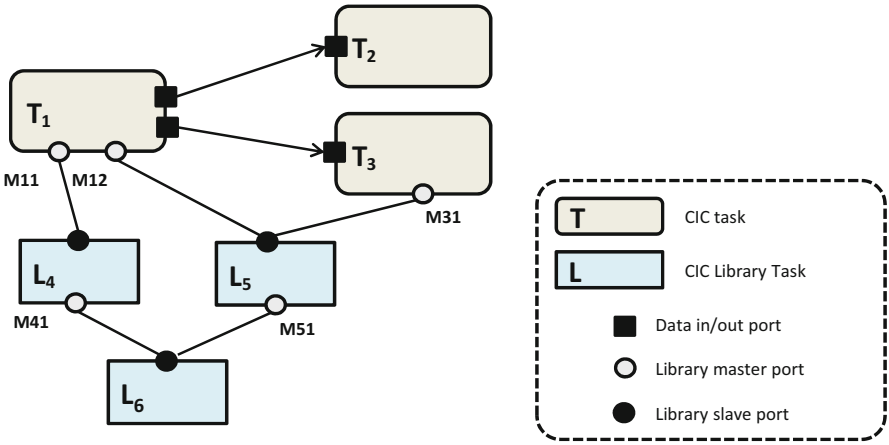


Fig. 29.7 An extended SDF graph that uses library tasks

shared resources in the SDF model. A library task is a sharable and mappable object that defines a set of function interfaces inside. Figure 29.7 shows an SDF graph that consists of three normal SDF tasks (T1–T3) and three library tasks (L4–L6). Connection with a library task is made between a pair of library ports, library master port and a library slave port that are represented by a white circle and a dark circle, respectively. A library task should have a single slave port that can be connected to multiple masters that share the library task. Since each library port has its own type that defines a set of function interfaces, connection between a master port and a slave port can be established only when their types are matched.

Unlike a normal SDF task, a library task is not invoked by input data but by a function call inside an SDF task; it is a passive object. There are specific rules to specify and use a library task in an SDF graph. Figure 29.8 illustrates code templates associated with a library task. A library task has two separate files associated: a library header file and a library code file. The library header file declares the library functions, while the library code file defines the function bodies. The prototype

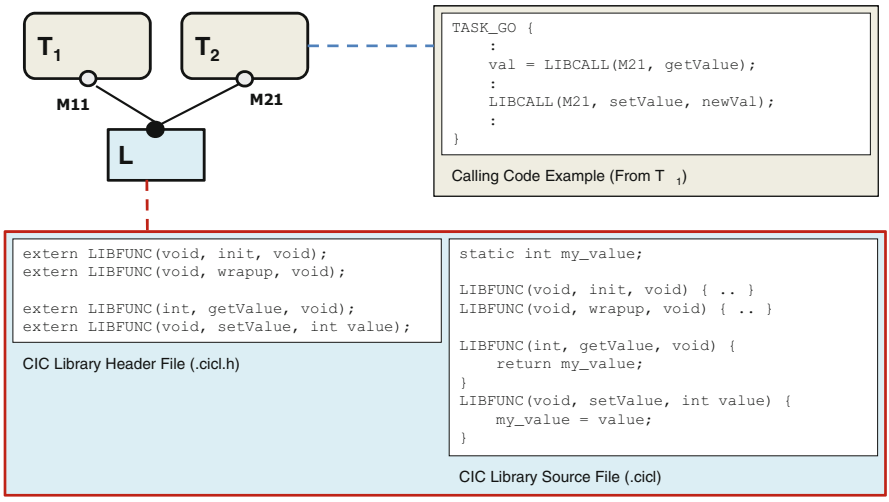


Fig. 29.8 Code templates associated with a library task

of a library function is defined by a directive, LIBFUNC(), that will be translated into a regular function definition automatically by the CIC translator. A library task should define *init* and *wrapup* functions like a normal SDF task for initialization and finalization of the library task.

A caller task uses LIBCALL directive to call a library function as shown in Fig. 29.8. The first parameter of LIBCALL() is the name of the library master port, the second is the function name, and the others are the arguments. If the function has a return value, it can be taken from the LIBCALL invocation. Note that pointers may not be used for arguments and return values to make the SDF graph portable to a variety of target architectures. For shared address space architectures, however, the developer may use pointers for efficient implementation, giving up portability.

A library task may have a persistent internal state, simply called a state. Then the access to the state should be protected by synchronization primitives, Lock() and Unlock() to avoid data race problems. In case multiple masters access the same library task that has a state, the return value of a library function may depend on the execution order of the master actors, which is anathema to any deterministic model. So, we explicitly specify a property of a library task whether it is deterministic or not. In case the library task has no state or returns the same value to the master tasks regardless of the calling order, the library task is classified as "deterministic." Otherwise, the developer should be aware that the library task does not guarantee deterministic behavior. Even though a library actor is nondeterministic in the sense that the return value to a master task depends on the scheduling order of master tasks, the same behavior can be repeated if the same scheduling order is followed.

There are several use cases of library task. A library task provides a way to share global variables or HW resources among multiple SDF tasks explicitly in a

systematic way. In a server-client application, the server task can be specified by a library task that may be shared by multiple clients. Note that we may change the number of clients arbitrarily since the number of master ports connected to a slave port can vary at run time. Another use case of a library task is to make a vertically layered software structure by providing a set of Application Programming Interfaces (APIs) of the software layer below the application layer.

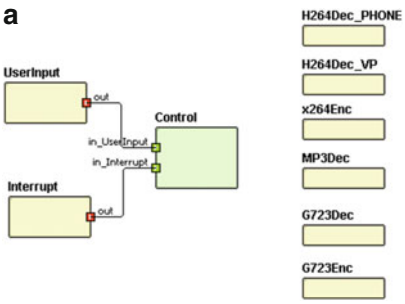
29.2.2 Dynamic Behavior Specification at the Top-Level Specification of the CIC Model

In this subsection, we explain how to specify the system-level dynamic behavior at the top level of CIC model. At the system level, the set of applications running concurrently may change or applications may change their operation modes according to user requests. Several approaches have been proposed to specify the dynamic behavior of data-flow applications. FunState that was proposed as an internal representation for codesign process [25] uses an FSM to control the activation of data-flow tasks. In STATEMATE [8], an extended FSM model, called statechart, specifies the entire system behavior and determines when to execute each task in the activity chart. Distributed application layer (DAL) [21] uses an FSM to add dynamism to distributed operation layer (DOL) [24] that is based on the KPN model. The FSM model of DAL specifies all use cases and how transitions between use cases occur, where a use case corresponds to a set of applications running concurrently, assuming that the number of use cases of the system is finite.

HOPES inherits the approach of its predecessor, PeaCE [7], where a control task is distinguished from application processes at the top level and plays the role of user-level supervisor that controls the execution status of applications. A control task uses an FSM model inside to specify the system-level dynamic behavior. Consider a simple smartphone example of Fig. 29.9. The system consists of two input processes running in the background, one control task, and six application processes. Each application is specified by an extended SDF graph inside; Fig. 29.5 is the internal specification of H.264 decoder application for instance. Suppose that there are four use cases, modes of operations, for the smartphone system as shown in Fig. 29.9b. In the Menu mode, there is no active application and the system waits for input events to be caught by two input processes, UserInput and Interrupt (phone arrival). Depending on the user input, the system changes the mode of operation and activates the associated applications. When a phone signal is detected, the system suspends the current mode of operation and switches its mode to the VideoPhone mode. After the call is completed, the system goes back to the suspended mode and resumes suspended applications.

The aforementioned description of the dynamic behavior is specified by an FSM inside the control task. Figure 29.10 shows the captured screen for the FSM specification in HOPES and the associated pseudocode automatically generated by the CIC translator. It has four states that correspond to four use cases. The default state is the *Menu* state, denoted by a bold circle. The control task is basically

a

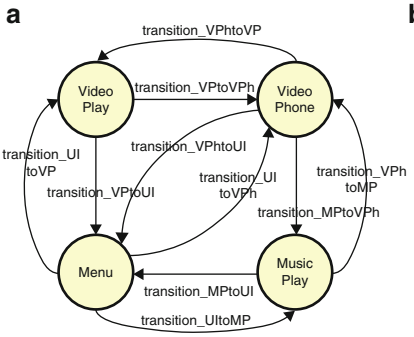


b

Use case	Active applications
Menu	-
VideoPhone	G.723 decoder, G.723 encoder, H.264 decoder (Phone), x.264 encoder
VideoPlay	MP3 decoder, H.264 decoder (VP)
MusicPlay	MP3 decoder

Fig. 29.9 (**a**) A simple smartphone example and (**b**) four use cases of the system

a



b

```
switch (current_state) {
case MENU:
   if(in_UserInput== 1)        next state = VideoPlay;
   else if(in_UserInput ==2)   next state = VideoPhone;
   else if(in_UserInput ==3)   next state = MusicPlay;
case VideoPlay:
   Execute VideoPlay or resume VideoPlay;
   if(in_Interrupt == On)
     Suspend VideoPlay & next state = VideoPhone;
   else if (terminated)
     next state = Menu;
case VideoPhone:
   Execute VideoPhone;
   if (terminated)
     if(previous_state == MusicPlay)
       next state = MusicPlay;
     else if(previous_state == VideoPlay)
       next state = VideoPlay;
…
} // end switch
```

Fig. 29.10 (**a**) FSM specification of the control task in the smartphone of Fig. 29.9 and (**b**) the pseudocode generated by the CIC translator

triggered by an event. There are three kinds of events. The first is an external event that is received from the input port of the control task, which is explicitly drawn at the top level of CIC model. The second kind is generated from the hidden supervisor internally by monitoring the execution status of applications. For instance, the system detects the termination of an application and generates an internal event. The last is a timeout event. The CIC control task can initiate a timer at a certain state. When the specified time is expired, a timeout event is generated by a timer that is another hidden component assumed in HOPES.

At each state, the programmer may use APIs to define the control action, which is similar to action scripts of the statechart in STATEMATE. The control APIs currently defined in HOPES are listed in Table 29.1. The first category is to control the execution status of an application and the second category is to change or monitor a specific parameter of an application. The third category is defined to explicitly specify the timing requirements of the system, and the last category controls the timer modules that are assumed to exist in the system. Since timing correctness is as important as value correctness in system functionality, explicit specification of timing requirement has been recently advocated for real-time

Table 29.1 Control APIs currently defined in HOPES

Category	APIs	Description
Execution status	SYS_REQ({RUN/STOP/SUSPEND/ RESUME} _TASK, task_name)	Run/stop/suspend/resume a task
	status = SYS_REQ(CHECK_TASK_STATE, take_name)	Check the execution status
Parameter control	SYS_REQ(SET_PARAM_{INT/FLOAT}, task_name, param_name, value)	Change the parameter value
	p_value = SYS_REQ(GET_PARAM_{INT /FLOAT}, task_name, param_name)	Get the parameter value
Timing requirement	SYS_REQ(SET_THROUGHPUT, task_name, thr _val)	Set the throughput requirement
	SYS_REQ(SET_DEADLINE, task_name, lat_val , lat_unit)	Set the latency requirement
Timer control	time_base = SYS_REQ(GET_CURRENT_TIME _BASE)	Get the current system time
	timer_id = SYS_REQ(SET_TIMER, time_base, offset)	Set timer to time_base + offset
	ret = SYS_REQ(GET_TIMER_ALARMED, timer_id)	Check if the timer is expired
	SYS_REQ(RESET_TIMER, timer_id)	Reset the timer

embedded system design. While PTIDES [4] uses a discrete event model of computation for timing specification, HOPES uses timing control APIs as a part of control task specification. We may initiate a timer and read the timer. In addition, we may set up the throughput or deadline requirement of an application. Note that the timing requirement of an application may change depending on the use cases. Those timing constraints are referred to in the design space exploration step when constructing the static schedule of an MTM-SDF graph.

29.3 Design Space Exploration in HOPES

As overviewed in Fig. 29.2, HOPES uses a Y-chart approach [13] to explore the design space by mapping applications to candidate architectures with a given set of objectives. Since the dynamic behavior of a system is not predictable, it is challenging to make a mapping decision and evaluate the decision. We have developed a novel hybrid mapping technique that combines compile-time static mapping of applications and run-time dynamic mapping of applications to available resources. Remind that each application is specified by an MTM-SDF graph so that each mode of operation can be statically scheduled. When we schedule each mode of

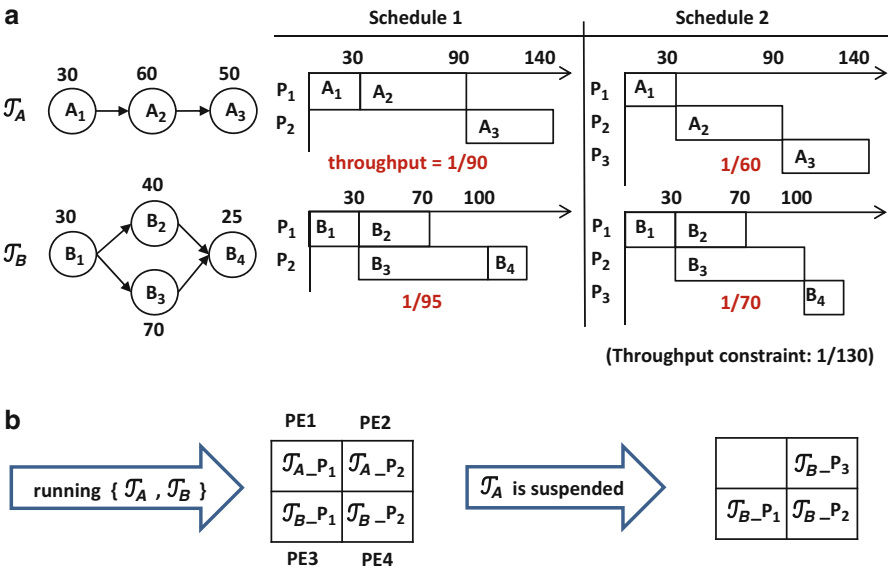


Fig. 29.11 An example of hybrid mapping: (**a**) Pareto-optimal mapping solutions of two applications, and (**b**) dynamic mapping results according to a given scenario of system status change

an application, we find a Pareto-optimal set of mapping decisions for each candidate architecture. Suppose that we have multiple objectives of mapping, minimizing the resource usage and maximizing the throughput performance for instance. Then static scheduling is performed for each application independently to obtain a set of Pareto-optimal solutions for multiple objectives. We assume that no processor sharing is allowed, or a processing element is dedicated to an application, in the current implementation of HOPES.

Figure 29.11 shows a simple example that consists of two applications, each of which has a single mode of operation. For each application, two Pareto-optimal mappings are found with varying number of processing elements. At run time, dynamic mapping is performed by first identifying which applications are running concurrently and next allocating the processing elements to the applications based on their Pareto-optimal mapping solutions. In this example, four processing elements are equally allocated to two applications, two to each. When application *A* is suspended, we reallocate the processing elements to the remaining application, *B*, to improve the throughput performance, which is also a Pareto-optimal mapping of *B*.

Dynamic remapping is triggered at every system status change. Some causes of the system status change are explicitly specified in the CIC model. For instance, the change of execution status or QoS requirement of an application is specified by a state transition defined in a control task. Thus, such a state transition triggers dynamic mapping. The operation mode change of an application is explicitly

specified in the MTM-SDF model. There are other causes, however, that are not specified in the CIC model. An example is the failure of a processing element. If a processor failure is detected, remapping of applications is performed [16]. We assume that all system status changes are captured by the hidden supervisor whatever the causes are.

29.3.1 Static Scheduling Technique of an MTM-SDF Graph

Since an application is specified by an MTM-SDF graph, we devised a novel static scheduling technique of an MTM-SDF graph. An important objective is to minimize the mode change overhead that may affect the real-time performance of an application. Figure 29.12 shows a simple MTM-SDF graph that has two modes of operation. When we use a naive technique that schedules each SDF graph independently, we need to migrate three tasks when mode change occurs as illustrated in Fig. 29.12b. It is better to consider the migration overhead when finding a static schedule for each mode. Another extreme approach is to avoid task migration by considering all modes simultaneously; this approach is assumed in the previous work [22]. As shown in Fig. 29.12c, this approach maps a task to the same processor at all modes and so requires more processors. The proposed approach is

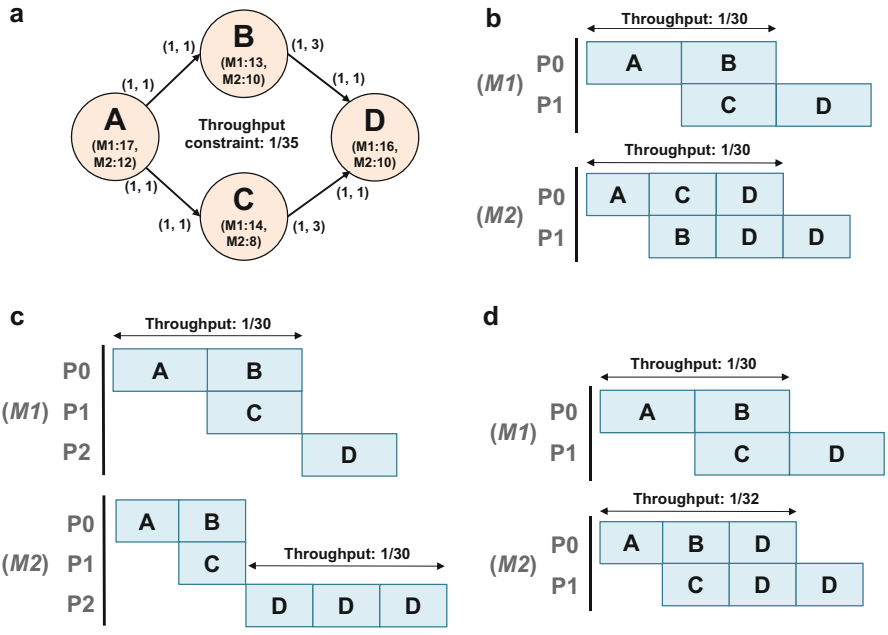
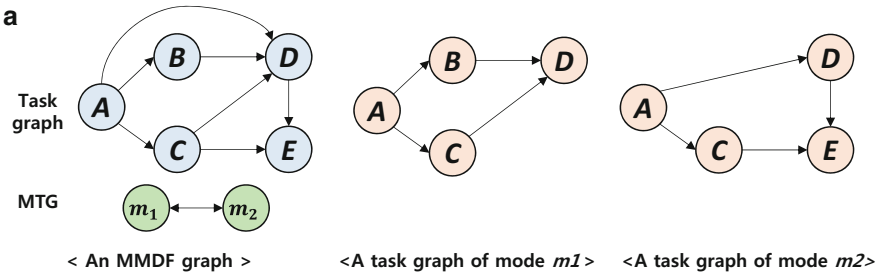
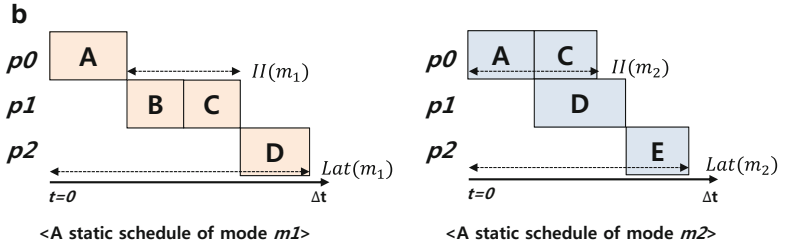


Fig. 29.12 (**a**) An MTM-SDF graph example and scheduling results for three cases: (**b**) scheduling each mode independently, (**c**) scheduling all modes simultaneously disallowing task migration, and (**d**) scheduling all modes simultaneously with task migration to minimize the resource requirement



An MMDF graph example



Static scheduling results of mode m_1 and m_2

Fig. 29.13 An MMDF graph example with two modes of operation and their static scheduling results. (**a**) An MMDF graph example taken from [11]. (**b**) Static scheduling results of mode m_1 and m_2

to consider all modes simultaneously, but allowing task migration to minimize the resource requirement, which results in the schedule of Fig. 29.12d.

How to consider the task migration overhead is the key challenge in the static scheduling of an MTM-SDF graph. Figure 29.13a shows a simple MMDF graph example that consists of two modes of operation. For each mode, a static schedule which satisfies the given throughput constraint is constructed as shown in Fig. 29.13b. If the schedule of each mode is repeated forever, the output samples will be produced periodically. The period is equal to the inverse of the throughput performance, which is denoted as the initiation interval (II) in the figure. Even though the static schedule of each mode satisfies the given throughput constraint, the overall throughput performance of the MMDF graph may not satisfy the throughput constraint because of the mode transition delay if a mode transition occurs.

The mode transition delay between two modes is defined how the time interval between the last output production time of the previous mode and the first output production time of the next mode is larger than the initiation interval of the next mode. Suppose that the last iteration of the previous mode is started at $t = 0$. First we formulate the start offset (χ) of the first iteration of the next mode. The start offset (χ) is determined by the following three factors:

(1) *Scheduling delay (D_{sched})*: To keep the temporal property of the given static schedule, we need to shift the start time of the subsequent mode. The time

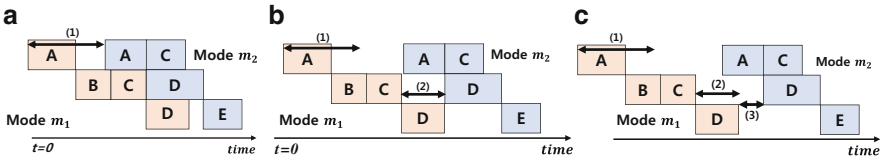


Fig. 29.14 Mode transition delay between static schedules of modes m_1 and m_2 in Fig. 29.13b. (**a**) Scheduling delay. (**b**) Task ordering delay. (**c**) Task migration delay, taken from [11]

interval, denoted by (1) in Fig. 29.14a, illustrates the scheduling delay between modes m_1 and m_2 of Fig. 29.13.

(2) *Task ordering delay* (D_{order}): Because the proposed technique allows task migration between modes, a task can be mapped onto different processors in each mode. So it needs to be guaranteed that two consecutive executions of the same task are not overlapped or inverted during mode change. In Fig. 29.14a, two executions of task D are overlapped between modes. Thus, the execution of the next mode should be delayed by the task ordering delay denoted by (2) in Fig. 29.14b.

(3) *Task migration delay* (D_{mig}): Tasks which are mapped onto different processors between modes should be migrated during the time interval between the end time in the previous mode and the start time in the next mode. If the time interval is not long enough to migrate the task, additional time delay is required. In Fig. 29.14b, task D should be migrated to another processor after the end of execution in the previous mode, and additional time delay is needed, which is the task migration delay denoted by (3) in Fig. 29.14c. In case of task C, no additional time delay is required.

Summing up all three types of delay mentioned above, we compute the start offset of the next mode as follows:

Definition 1 (Start offset of mode m in the case of mode transition $n \rightarrow m$).

$$\chi^{nm} = D^{nm}_{sched} + D^{nm}_{order} + D^{nm}_{mig}$$

Since the output production time of each mode equals to the latency of the static schedule from the start time, the mode transition delay can be formulated as follows:

Definition 2 (Mode transition delay from mode n to mode m).

$$TransDelay(n,m) = Lat(m) + \chi^{nm} - Lat(n) - II(m)$$

where $Lat(m)$ represents the latency of mode m and $II(m)$ represents the initiation interval of mode m.

Note that, if $Lat(m) + \chi^{nm} - Lat(n) \leq II(m)$, then $TransDelay(n, m)$ will be smaller than zero. It means that the time interval of the output production times during a mode transition can be shorter than the output production time interval of the next mode ($II(m)$).

The mode transition delay will be used to determine the new throughput requirement for each mode of operation to satisfy the throughput constraint. When the number of iterations performed in mode m is N, the average initiation interval becomes $MaxTransDelay(m) + N * II(m)/N$ where $MaxTransDelay(m)$ indicates the maximum value of all possible mode transitions to mode m. Therefore, we need to increase the throughput performance by decreasing $II(m)$, in order to satisfy the given throughput requirement. In other words, the new initiation interval $II_{new}(m)$, whose inverse is the new throughput requirement, should satisfy the following inequality.

$$MaxTransDelay(m) + N * II_{new}(m) \leq N * 1/(throughput\ requirement) \tag{29.1}$$

When we schedule all modes of MTM-SDF graphs, we have to consider the increase of throughput requirement for each possible pair of mode changes. The proposed scheduling technique is based on a Genetic Algorithm (GA) [18], of which the overall procedure is shown in Fig. 29.15. The chromosome for GA represents which processor a task in each execution mode is mapped. Chromosomes of initial population are randomly generated and selected from crossover and mutation. The probabilities of crossover and mutation are given by a user with configuration parameters. In the local optimization step, we shuffle the processor indexes for some

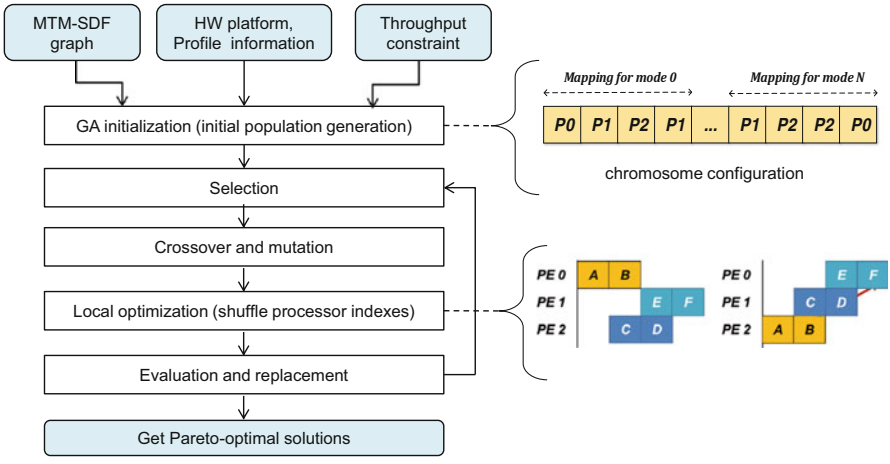


Fig. 29.15 GA-based MTM-SDF scheduling framework in HOPES

selected modes in the chromosome in case shuffling reduces the migration cost at mode change. Note that a hardware component is regarded as a special processing element to which a limited set of tasks can be mapped.

In the evaluation step, we apply a list scheduling heuristic to find a static schedule based on the mapping information of the chromosome. Once we construct a static schedule, we evaluate the fitness value of each solution and check whether the throughput constraint is satisfied or not. Chromosomes in the population are sorted by their fitness value and poor chromosomes are eliminated.

For each Pareto-optimal solution, we record the mapping and scheduling result of tasks for a given set of processing elements. And we determine the minimum buffer size for each channel by finding out the maximum number of samples accumulated on the channel during an iteration of the schedule. Note that we may expand the design space by considering the variation of voltage and frequency for power minimization, which has not been implemented in HOPES yet.

29.3.2 Dynamic Mapping

Actual mapping of tasks to processors is performed at run time based on the scheduling information of all applications. When a system status change is detected, the supervisor identifies the set of applications concurrently running and the set of available processors. And it allocates the processors to applications in order to maximize the overall Quality of Service (QoS) metric.

Run-time dynamic mapping is performed in two steps: processor allocation and processor binding. In the processor allocation step, we determine the number of processors allocated to each application. We first allocate the minimum number of processors to each application in order to satisfy the throughput constraint. If the sum of allocated processors is larger than the number of available processors, all applications are not schedulable and we have to discard some applications of low criticality. If there are remaining processors, we repeat the following process until there are no remaining processors or no gain is expected with more processors allocated to any application: find an application that would have the maximum benefit with one more processor and allocate a remaining processor to the application. Suppose that the number of available processors is five in the example of Fig. 29.11. After allocating two processors to both applications initially to satisfy the throughput constraints, one processor is left unallocated. Since the throughput improvement of application A with one more processor is larger than that of application B, we allocate the remaining processor to application A.

After processor allocation is finished, we perform the processor binding step where the physical position of the allocated processors is determined. A popular objective of the binding step is to minimize the average communication overhead over all applications and to minimize the task migration overhead. To minimize the task migration overhead, the same binding is preserved for an application that has no change in the number of allocated processors.

29.4 CIC Translator: Automatic Code Synthesis from the CIC Model

A key benefit of the proposed model-based design methodology is that the target code can be synthesized automatically from the CIC model after the mapping and scheduling decision is made for a given HW/SW platform. The code synthesis step can be understood as model refinement, enjoying the benefit of "correct-by-construction" design paradigm to relieve the designer of heavy burden of verifying the correctness that can be checked by static analysis of the model. To this end, the code should be synthesized in a way to preserve the interface and execution semantics of the model. For an SDF task, for instance, it should be guaranteed that the task starts its execution only after all input ports have as many number of samples as are defined by the sample rates on the associated channels. It implies that we may need to synthesize an interface module in front of the HW IP to synchronize the arrival of input data samples if an SDF task is implemented by a HW IP. Even though the SDF model assumes infinite size of channel buffers, we can determine the buffer sizes at compile time from the static analysis. Then an SDF task should check before starting its execution if there is available space at the output buffers.

In HOPES, we assume that the internal code of an SDF task is given. It is up to the designer to guarantee the correctness of the internal code. Then the CIC translator synthesizes the interface code between tasks and the scheduler code to determine the execution order of the mapped tasks on each processing element. The interface code and the scheduler code depend on the mapping and scheduling policy of the target platform. There are four policies to perform mapping and scheduling of SDF tasks: fully static, self-timed, static assignment, and fully dynamic. If the fully static policy is applied, the run-time scheduler keeps not only the mapping and scheduling decision made at compile time but also the timing information. If a task finishes earlier than the worst-case execution assumed in static scheduling, the run-time scheduler delays the completion of the task until the assumed completion time. By keeping the start and the completion time of tasks, the fully static policy guarantees to produce the same scheduling result as expected at compile time. It means that real-time performance is guaranteed to be correct by construction, which is very desirable for hard real-time systems. The main drawback of this policy is that we should sacrifice the processor utilization in case the worst-case task execution scenario is very different from the average-case scenario. The run-time scheduler simply executes the tasks at the predetermined starting times without checking the buffer status.

Under the self-timed policy, on the other hand, the run-time scheduler does not keep the starting times of tasks while preserving the mapping and scheduling result. Before it starts the next task on the schedule list, it should check the availability of the input data samples. Since the scheduling order is preserved, we may generate a single thread that executes a sequence of function calls in the scheduling order where each SDF task is implemented as a function call. Note that we do not resort to any OS scheduler of the target platform under this policy.

The static assignment policy allows the change of task execution order while the mapping decision is kept. In a self-timed policy, a processor can be idle waiting for the arrival of input samples for the next task to execute in the scheduling order even though there is an executable task. By changing the scheduling order at run time, we may increase the processor utilization, which is the main reason of adopting a static assignment policy particularly when the task execution times vary widely. The static scheduling information can be used to assign the priority of the tasks, giving a higher priority to the task that appears earlier in the scheduling order. If a static assignment policy is used, we synthesize each task as a separate thread and may resort to the thread scheduler that is provided by the SW platform of the target architecture. If there is no built-in thread scheduler, we synthesize a simple run-time scheduler that checks the execution status of all tasks when the processor receives a data sample from the other processors and completes the execution of the current thread. Thus, there is a trade-off between run-time scheduling overhead and processor utilization between self-timed and static assignment policy.

The fully dynamic policy ignores the static scheduling information at run time by allowing the change of mapping and scheduling of tasks. It is the same as the global scheduling policy for an Symmetric Multi-Processing (SMP) processor, distinguished from the partitioned scheduling policy where mapping of tasks does not change at run time. Similarly to a static assignment policy, we may use the static scheduling information to assign the priority of the tasks. And we synthesize each task as a separate thread and use the global scheduler that is provided by the SW platform of the target architecture. In the current implementation, the fully dynamic policy can be used for an SMP target only.

Figure 29.16a shows the overall flow of automatic code synthesis by the CIC translator, and Fig. 29.16b shows the structure of the synthesized code. We use colors to show how the synthesized code is matched with the synthesis flow in the figure. We first check which mapping and scheduling policy will be used and

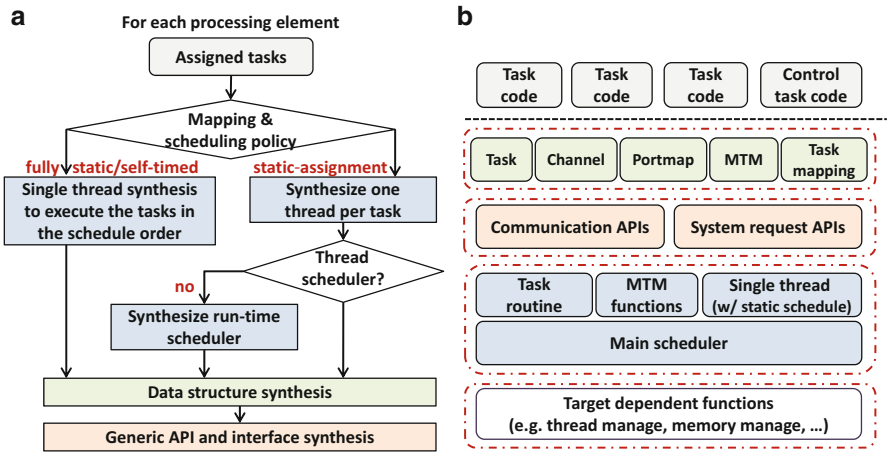


Fig. 29.16 (**a**) The overall flow of automatic code synthesis by the CIC translator and (**b**) the structure of the synthesized code

partitions the tasks based on the mapping information unless the fully dynamic policy is used. Then the target code for each processing element is synthesized one by one. In case the fully static or the self-timed policy is used for a processor, we synthesize a single thread that executes the mapped tasks by function calls following the scheduling order. In case the static assignment policy is used, a separate thread is created for each task and a run-time scheduler code is synthesized if there is no built-in thread scheduler in the SW platform. Depending on the policy and the SW platform, we translate the generic APIs to the target APIs when the task code is synthesized. Since the interface code with the other processing elements is dependent on the target platform, we assume that the interface code is given as a part of input information to the CIC translator. To this end, HOPES has target-specific library folders that contain target-specific tasks as well.

The CIC translator can be understood as a high-level compiler of the CIC model to generate the target-specific code automatically. As we need a different C compiler to generate the target-specific binary from a target-independent C code, we need to develop a different CIC translator for each target platform. As of now, the following target platforms are supported in the HOPES environment: Linux-based SMP processor, CPU-GPU heterogeneous architecture, IBM Cell processor, Network-on-Chip (NoC)-based many-core virtual prototype, and a multi-robot platform with Bluetooth communication links.

29.5 Experimental Results

In this section, we show two real-life examples to demonstrate the overall design flow to verify the viability of the HOPES methodology. The first example is a smartphone example shown in Fig. 29.9. This example is quite challenging since it consists of multiple applications running concurrently, having inter-application and intra-application dynamism. And each application has real-time constraints. The profiling information of applications is obtained by preparatory experiments in advance using a cycle-accurate ARM processor simulator. The WCET information for each task is reported in Table 29.2 for the H.264 decoder application of Fig. 29.5. Tables 29.3 and 29.4 show the WCET of each task in x264 encoder and MP3 decoder application. Both applications are specified with an SDF graph respectively, having only one mode of operation; the tables show how many tasks each application consists of. For the x264 encoder application, we make a single task for the most time-consuming algorithm, motion estimation (ME), in this experiment. G.723 encoder and G.723 decoder applications are specified by a single task each, and their execution times are profiled to 4×10^3 cycles/iteration and 6×10^3 cycles/iteration, respectively.

With the given profiling information, compile-time analysis is performed to obtain the set of Pareto-optimal mapping and scheduling solutions for varying number of processors for each application. The result of compile-time analysis is summarized in Table 29.5.

To compare the performance of the proposed hybrid mapping technique with a dynamic mapping technique, we tested the following scenario: (1) play a video clip,

Table 29.2 Profiling information of H.264 decoder application (unit: $\times 10^3$ cycles/frame)

Task	Time (WCET/ average)	Task	Time(WCET/average)	Task	Time (WCET/ average)
ReadFile	I: 980/760 P: 590/420	IntraPredY	I: 980/830	InterPredY	I: 80/60 P: 3940/1560
Decode	I: 7500/5010 P: 2990/920	IntraPredU	I: 190/150	InterPredU	I: 20/20 P: 340/270
Deblock	I: 1550/1390 P: 1120/370	IntraPredV	I: 180/150	InterPredV	I: 20/20 P: 340/270
WriteFile	I: 2240/2120 P: 2360/2100				

Table 29.3 Profiling information of x264 encoder application (unit: $\times 10^3$ cycles/frame)

Task	Time (WCET/average)	Task	Time (WCET/average)	Task	Time (WCET/average)
Init	250/170	Deblock	3020/2660	Encoder	4840/4470
ME	15170/14720	VLC	2350/1780		

Table 29.4 Profiling information of MP3 decoder application (unit: $\times 10^3$ cycles/iteraion)

Task	Time (WCET/average)	Task	Time (WCET/average)	Task	Time (WCET/average)
VLD	810/150	Antialias	40/10	Stereo	70/30
DeQ	690/300	Hybrid	1230/160	Reorder	50/10
Subband	630/270	WriteFile	150/20		

Table 29.5 Summary of compile-time analysis

Application	Processors (min, max)	Throughput (min, max)	Throughput constraint
H.264 decoder	(1,2)	(50.9, 52.8) frames/sec	VideoPlay: 30 frames/sec VideoPhone: 15 frames/sec
MP3 decoder	(1,6)	(123.7, 569.1) iterations/sec	150 iterations/sec
x264 encoder	(1,2)	(27.3, 30.1) frames/sec	15 frames/sec

(2) a phone call preempts the video play, (3) resume the video play after the call is finished, and (4) return to the Menu state when the video clip is finished. We assume that the target HW platform has a 3×3 NoC architecture in which there are seven ARM processor tiles (700 Mhz for each) available for executing the applications. Figure 29.17a shows the total sum of throughput excesses over the throughput constraints for all applications. The throughput excess can be used to reduce the power consumption of the system by lowering the voltage and frequency of the processor. Figure 29.17b illustrates the relative latency achieved from the dynamic mapping against the proposed hybrid mapping, varying the communication-to-

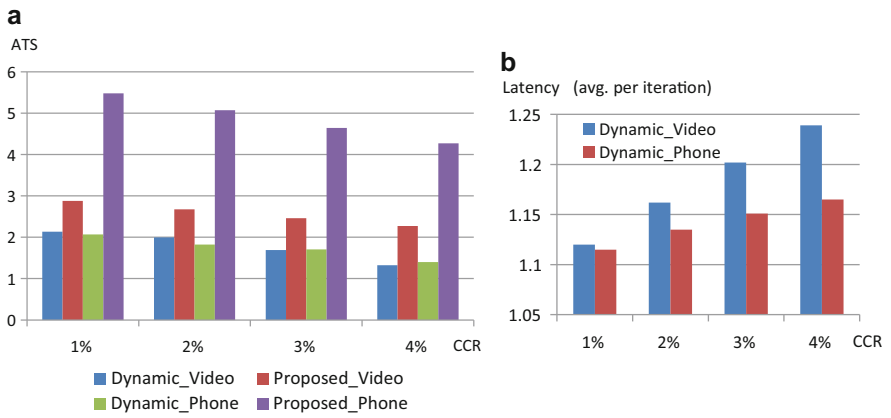


Fig. 29.17 (**a**) The aggregate throughput gain over the throughput constraints for both hybrid and dynamic mapping techniques, and (**b**) the relative latency achieved by dynamic mapping against the hybrid mapping

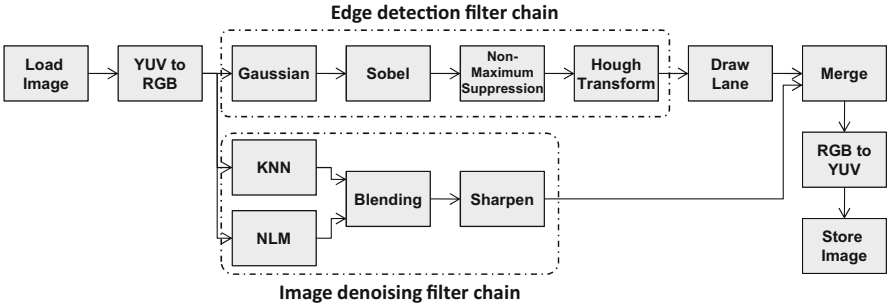


Fig. 29.18 A CIC modeling of lane detection algorithm

computation ratio (*ccr*). These experiments confirm that the hybrid mapping gives significant gain in throughput and latency by utilizing the static scheduling results.

As another real example, a lane detection algorithm for driver assistance is implemented by a CPU-GPU heterogeneous architecture. In this experiment, we used Intel Core i7-930 CPU (2.80 GHz) and two Tesla M2050 GPUs. To run a task on a GPU, we used a different version of the task that uses CUDA programming in its internal definition. For CUDA programming, NVIDIA GPU Computing SDK 3.1 and CUDA toolkit v3.2 RC2 were used. The CIC model of the lane detection application is displayed in Fig. 29.18 and the associated profiling information is shown in Table 29.6.

The design space explored in this experiment is defined by the number of CPU and GPU processing elements, task mapping, and communication methods between CPU and GPU. Asynchronous communication between CPU and GPU is supported by defining *stream*s in CUDA programming. While operations with the same stream

Table 29.6 Profiling information of lane detection application (unit: usec)

Task	CPU	GPU	Task	CPU	GPU
LoadImage	479	–	KNN	2,999,704	7202
YUVtoRGB	53,111	8152	NLM	1,017,401	16,497
Gaussian	78,100	4591	Blending	16,093	5078
Sobel	10,041	5139	Sharpen	110,139	5455
Non-max	164,013	6611	Merge	32,340	5032
Hough	311,966	5653	RGBtoYUV	66,733	4888
Draw lane	1592	–	StoreImage	1068	-

Table 29.7 Design space exploration of lane detection application (unit:sec)

Configuration	Sync	Async (2 streams)	Async (3 streams)	Async (4 streams)
CPU + 0 GPU	2109.5	–	–	–
CPU + 1 GPU	15.0	12.0	12.3	12.1
CPU + 2 GPUs	11.3	10.2	9.8	9.8

Table 29.8 Task mapping onto 1 CPU + 2 GPUs

Processor	Tasks
CPU	LoadImage, Draw lane, StoreImage
GPU 0	YUVtoRGB, Gaussian, Sobel, Non-maximum, Hough, Merge
GPU 1	KNN, NLM, Blending, Sharpen, RGBtoYUV

should be serialized, those between different streams can be executed in parallel. Thus, asynchronous communication promises potential throughput improvement paying the overhead of memory space and stream management overhead. For this experiment, we used a yuv video clip which consists of 300 frames of HD size (1280×720). We explored the design space manually to obtain the result as shown in Table 29.7. It reveals that using two GPUs gives the best performance in which task mapping is made as shown in Table 29.8.

29.6 Current Status and Conclusion

The HOPES design environment consists of various tools that realize individual design steps in the design flow of Fig. 29.2. It has an eclipse-based Graphical User Interface (GUI) to help a designer to follow the design flow conveniently. Interface between design tools is made by *xml* files so that we may change or add a design tool into the environment by accessing the interface files. We expect that the HOPES environment can be improved by third-party tools.

Besides the techniques introduced in this chapter, there are other tools involved in the HOPES design environment such as Worst-Case Response Time (WCRT) analysis tool (STBA and HPA) [14] and a HW/SW cosimulation tool, HSIM [26]. The WCRT analysis tool is to estimate the latency of an application conservatively

when a self-timed or a static assignment policy is adopted. Since the scheduling anomaly may happen due to unexpected interference from the other processing elements in the access to the shared resources, the worst-case performance estimated from the static analysis step is not guaranteed if we change the scheduling times of tasks or the execution order of tasks. Therefore, we use a separate tool to estimate the response of an application after mapping decision is made. The HW/SW cosimulation tool is used to run the target software without the real hardware platform.

In this chapter, it is confirmed that the HOPES methodology is viable to design complex real-time embedded systems with two real-life examples. But it is still far from a general system-level design tool to be used in practice and there is much room for improvement. First of all, we need to consider more real-life systems with diverse characteristics in the system behavior and the target architecture, which is not an easy job for academia. We are testing various types of hardware platforms, including Intel Xeon-Phi, IBM cell processor, many-core simulator, and cooperating heterogeneous robot platforms. The most time-consuming is to make a CIC translator for each target platform. It is similar to building a new C compiler for a new processor. Since the quality of design depends on the CIC translator, generating a target code is not sufficient for practical use. We have to synthesize as good quality code as a manually written code. Since the HOPES framework starts with CIC specification of an application, it is necessary to translate the legacy code to the CIC model for the reuse of a legacy code. If the legacy code is small enough to compose a single CIC task, translation could be made easily by modifying the interface code with the outside. Otherwise, it is necessary to restructure the legacy code to partition it to a set of CIC tasks that follow the assumed execution model, which should be done manually.

Even though the CIC model is independent of the target architecture, we may need to define target-dependent tasks. For instance, a task that accesses I/O devices usually needs to use OS-dependent APIs. To run a task on a special processing element, such as GPU and hardware IP, we need to have multiple versions of the same task that are dependent on the target architecture. Since the granularity of a task is large, careful consideration needs to be made to make it target-independent.

References

1. Bhattacharya B, Bhattacharyya SS (2001) Parameterized dataflow modeling for DSP systems. IEEE Trans Signal Process 49(10):2408–2421. doi:10.1109/78.950795
2. Buck JT (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Technical report, Department of EECS, UC Berkeley, Berkeley. Technical report UCB/ERL 93/69, Ph.D dissertation
3. Buck JT, Ha S, Lee EA, Messerschmitt DG (1994) Ptolemy: a framework for simulating and prototyping heterogenous systems. Int J Comput Simul 4(2):155–182
4. Eidson J, Lee EA, Matic Slobodan SSA, Zou J (2012) Distributed real-time software for cyber-physical systems. Proc IEEE 100(1):45-59

5. Girault A, Lee B, Lee E (1999) Hierarchical finite state machines with multiple concurrency models. IEEE Trans Comput Aided Des Integr Circuits Syst 18(6):742–760
6. Goossens S, Akesson B, Koedam M, Nejad AB, Nelson A, Goossens K (2013) The CompSOC design flow for virtual execution platforms. In: Proceedings of the 10th FPGAworld conference. ACM, p 7
7. Ha S, Kim S, Lee C, Yi Y, Kwon S, Joo YP (2008) Peace: a hardware-software codesign environment for multimedia embedded systems. ACM Trans Des Autom Electron Syst 12(3):24:1–24:25. doi:10.1145/1255456.1255461
8. Harel D, Naamad A (1996) The STATEMATE semantics of statecharts. ACM Trans Softw Eng Methodol (TOSEM) 5(4):293–333
9. Haubelt C, Falk J, Keinert J, Schlichter T, Streubühr M, Deyhle A, Hadert A, Teich J (2007) A SystemC-based design methodology for digital signal processing systems. EURASIP J Embed Syst 2007(1):1–22. doi:10.1155/2007/47580
10. Jung H, Lee C, Kang SH, Kim S, Oh H, Ha S (2014) Dynamic behavior specification and dynamic mapping for real-time embedded systems: HOPES approach. ACM Trans Embed Comput Syst (TECS) 13:135:1–135:26
11. Jung H, Oh H, Ha S (2017) Multiprocessor scheduling of a multi-mode dataflow graph considering mode transition delay. ACM Trans. Des. Autom. Electron. Syst. (TODAES) 22, 2, Article 37
12. Kangas T, Kukkala P, Orsila H, Salminen E, Hännikäinen M, Hämäläinen TD, Riihimäki J, Kuusilinna K (2006) Uml-based multiprocessor soc design framework. ACM Trans Embed Comput Syst 5(2):281–320. doi:10.1145/1151074.1151077
13. Kienhuis B, Deprettere E, Vissers K, Wolf PVD (1997) An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings of the IEEE international conference on application-specific systems, architectures and processors, pp 338–349. doi:10.1109/ASAP.1997.606839
14. Kim J, Oh H, Choi J, Ha H, Ha S (2013) A novel analytical method for worst case response time estimation of distributed embedded systems. In: Proceedings of the design automation conference (DAC), Austin, pp 1–10
15. Kwon S, Kim Y, Jeun WC, Ha S, Paek Y (2008) A retargetable parallel programming framework for MPSoC. ACM Trans Des Autom Electron Syst (TODAES) 13:39:1–39:18
16. Lee C, Kim H, Park H, Kim S, Oh H, Ha S (2010) A task remapping technique for reliable multi-core embedded systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), Scottsdale, pp 307–316
17. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
18. Man KF, Tang KS, Kwong S (1996) Genetic algorithms: concepts and applications [in engineering design]. IEEE Trans Ind Electron 43(5):519–534. doi:10.1109/41.538609
19. Nikolov H, Thompson M, Stefanov T, Pimentel A, Polstra S, Bose R, Zissulescu C, Deprettere E (2008) Daedalus: toward composable multimedia MP-SoC design. In: Proceedings of the design automation conference, pp 574–579
20. Park Hw, Jung H, Oh H, Ha S (2011) Library support in an actor-based parallel programming platform. IEEE Trans Ind Inf 7:340–353
21. Schor L, Bacivarov I, Rai D, Yang H, Kang SH, Thiele L (2012) Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Proceedings of the international conference on compilers architecture and synthesis for embedded systems (CASES), pp 71–80
22. Stuijk S, Geilen M, Theelen BD, Basten T (2011) Scenario-Aware dataflow: modeling, analysis and implementation of dynamic applications. In: Proceedings of the international conference on embedded computer systems: architectures, modeling, and simulation, ICSAMOS'11. IEEE Computer Society, pp 404–411. doi:10.1109/SAMOS.2011.6045491
23. Theelen BD, Geilen M, Basten T, Voeten J, Gheorghita SV, Stuijk S (2006) A Scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of international conference on formal methods and models for co-design, MEMOCODE'06. IEEE Computer Society, pp 185–194. doi:10.1109/MEMCOD.2006.1695924

24. Thiele L, Bacivarov I, Haid W, Huang K (2007) Mapping applications to tiled multiprocessor embedded systems. In: International conference on application of concurrency to system design, pp 29–40. doi:10.1109/ACSD.2007.53
25. Thiele L, Strehl K, Ziegenbein D, Ernst R, Teich J (1999) FunState–an internal design representation for codesign. In: White JK, Sentovich E (eds) ICCAD. IEEE, pp 558–565
26. Yun D, Kim S, Ha S (2012) A parallel simulation technique for multicore embedded systems and its performance analysis. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 31:121–131

DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips

30

Todor Stefanov, Andy Pimentel, and Hristo Nikolov

Abstract

The complexity of modern embedded systems, which are increasingly based on heterogeneous multiprocessor system-on-chip (MPSoC) architectures, has led to the emergence of system-level design. To cope with this design complexity, system-level design aims at raising the abstraction level of the design process from the register-transfer level (RTL) to the so-called electronic system level (ESL). However, this opens a large gap between deployed ESL models and RTL implementations of the MPSoC under design, known as the *implementation gap*. Therefore, in this chapter, we present the DAEDALUS methodology which the main objective is to bridge this implementation gap for the design of streaming embedded MPSoCs. DAEDALUS does so by providing an integrated and highly automated environment for application parallelization, system-level design space exploration, and system-level hardware/software synthesis and code generation.

Acronyms

ADG	Approximated Dependence Graph
CC	Communication Controller
CM	Communication Memory
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DSE	Design Space Exploration
DWT	Discrete Wavelet Transform
ESL	Electronic System Level
FCFS	First-Come First-Serve

T. Stefanov (✉) • H. Nikolov
Leiden University, Leiden, The Netherlands
e-mail: t.p.stefanov@liacs.leidenuniv.nl; h.n.nikolov@gmail.com

A. Pimentel
University of Amsterdam, Amsterdam, The Netherlands
e-mail: a.d.pimentel@uva.nl

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_30

FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
GCC	GNU Compiler Collection
GUI	Graphical User Interface
HW	Hardware
IP	Intellectual Property
IPM	Intellectual Property Module
ISA	Instruction-Set Architecture
JPEG	Joint Photographic Experts Group
KPN	Kahn Process Network
MIR	Medical Image Registration
MJPEG	Motion JPEG
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
OS	Operating System
PIP	Parametric Integer Programming
PN	Process Network
PPN	Polyhedral Process Network
RTL	Register Transfer Level
SANLP	Static Affine Nested Loop Program
STree	Schedule Tree
SW	Software
UART	Universal Asynchronous Receiver/Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XML	Extensible Markup Language
YML	Y-chart Modeling Language

Contents

30.1 Introduction

The complexity of modern embedded systems, which are increasingly based on heterogeneous multiprocessor system-on-chip (MPSoC) architectures, has led to the emergence of system-level design. To cope with this design complexity, system-level design aims at raising the abstraction level of the design process to the so-called electronic system level (ESL) [18]. Key enablers to this end are, for example, the use of architectural platforms to facilitate reuse of IP components and the notion of high-level system modeling and simulation [21]. The latter allows for capturing the behavior of platform components and their interactions at a high level of abstraction. As such, these high-level models minimize the modeling effort and are optimized for execution speed and can therefore be applied during the very early design stages to perform, for example, architectural design space exploration (DSE). Such early DSE is of paramount importance as early design choices heavily influence the success or failure of the final product.

System-level design for MPSoC-based embedded systems typically involves a number of challenging tasks. For example, applications need to be decomposed into parallel specifications so that they can be mapped onto an MPSoC architecture [29]. Subsequently, applications need to be partitioned into hardware (HW) and software (SW) parts because MPSoC architectures often are heterogeneous in nature. To this end, MPSoC platform architectures need to be modeled and simulated at ESL level of abstraction to study system behavior and to evaluate a variety of different design options. Once a good candidate architecture has been found, it needs to be synthesized. This involves the refinement/conversion of its architectural components from ESL to RTL level of abstraction as well as the mapping of applications onto the architecture. To accomplish all of these tasks, a range of different tools and tool-flows is often needed, potentially leaving designers with all kinds of interoperability problems. Moreover, there typically exists a large gap between the deployed ESL models and the RTL implementations of the system under study, known as the *implementation gap* [32, 37]. Therefore, designers need mature methodologies, techniques, and tools to effectively and efficiently convert ESL system specifications to RTL specifications.

In this chapter, we present the DAEDALUS methodology [27, 37, 38, 40, 51] and its techniques and tools which address the system-level design challenges mentioned above. The DAEDALUS main objective is to bridge the aforementioned implementation gap for the design of streaming embedded MPSoCs. The main idea is, starting with a functional specification of an application and a library of predefined and pre-verified IP components, to derive an ESL specification of an MPSoC and to refine and translate it to a lower RTL specification in a systematic

and automated way. DAEDALUS does so by providing an integrated and highly automated environment for application parallelization (Sect. 30.4), system-level DSE (Sect. 30.5), and system-level HW/SW synthesis and code generation (Sect. 30.6).

30.2 The DAEDALUS Methodology

In this section, we give an overview of the DAEDALUS methodology [27, 37, 38, 40, 51]. It is depicted in Fig. 30.1 as a design flow. The flow consists of three main design phases and uses specifications at four levels of abstraction, namely, at FUNCTIONAL-LEVEL, ESL, RTL, and GATE-LEVEL. A typical MPSoC design with DAEDALUS starts at the most abstract level, i.e., with a FUNCTIONAL-LEVEL specification which is an application written as a sequential *C* program representing the required MPSoC behavior. Then, in the first design phase, an ESL specification of the MPSoC is derived from this functional specification by (automated) application parallelization and automated system-level DSE. The derived ESL specification consists of three parts represented in XML format:

1. *Application specification*, describing the initial application in a parallel form as a set of communicating application tasks. For this purpose, we use the polyhedral process network (PPN) model of computation, i.e., a network of concurrent processes communicating via FIFO channels. More details about the PPN model are provided in Sect. 30.3;
2. *Platform specification*, describing the topology of the multiprocessor platform;
3. *Mapping specification*, describing the relation between all application tasks in *application specification* and all components in *platform specification*.

For applications written as parameterized static affine nested loop programs (SANLP) in *C*, a class of programs discussed in Sect. 30.4, PPN descriptions can be derived automatically by using the PNGEN tool [26, 56], see the top-right part in Fig. 30.1. Details about PNGEN are given in Sect. 30.4. By means of automated (polyhedral) transformations [49, 59], PNGEN is also capable of producing alternative input-output equivalent PPNs, in which, for example, the degree of parallelism can be varied. Such transformations enable functional-level design space exploration. In case the application does not fit in the class of programs, mentioned above, the PPN application specification at ESL needs to be derived by hand.

The platform and mapping specifications at ESL are generated automatically as a result of a system-level DSE by using the SESAME tool [8, 39, 42, 53], see the top-left part of Fig. 30.1. Details about SESAME are given in Sect. 30.5. The components in the platform specification are taken from a library of (generic) parameterized and predefined/verified IP components which constitute the platform model in the DAEDALUS methodology. Details about the platform model are given in Sect. 30.6.2. The platform model is a key part of the methodology because it allows alternative MPSoCs to be easily built by instantiating components, connecting them, and setting their parameters in an automated way. The components in the library are

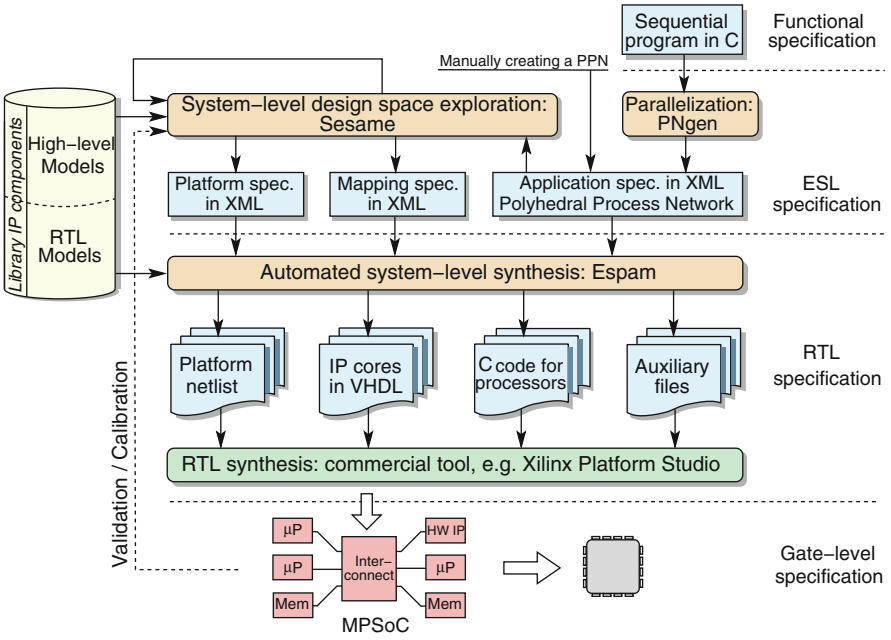


Fig. 30.1 The DAEDALUS design flow

represented at two levels of abstraction: high-level models are used for constructing and modeling multiprocessor platforms at ESL; low-level models of the components are used in the translation of the multiprocessor platforms to RTL specifications ready for final implementation. As input, SESAME uses the application specification at ESL (i.e., the PPN) and the high-level models of the components from the library. The output is a set of pairs, i.e., a platform specification and a mapping specification at ESL, where each pair represents a non-dominated mapping of the application onto a particular MPSoC in terms of performance, power, cost, etc.

In the second design phase, the ESL specification of the MPSoC is systematically refined and translated into an RTL specification by automated system-level HW/SW synthesis and code generation, see the middle part of Fig. 30.1. This is done in several steps by the ESPAM tool [25, 34, 36, 37]. Details about ESPAM are given in Sect. 30.6. As output, ESPAM delivers a hardware (e.g., synthesizable VHDL code) description of the MPSoC and software (e.g., C/C++) code to program each processor in the MPSoC. The hardware description, namely, an RTL specification of a multi-processor system, is a model that can adequately abstract and exploit the key features of a target physical platform at the register-transfer level of abstraction. It consists of two parts: *(1) platform topology*, a netlist description defining in greater detail the MPSoC topology and *(2) hardware descriptions of IP cores*, containing predefined and custom IP cores (processors, memories, etc.) used in *platform topology* and selected from the library of IP components.

Also, ESPAM generates custom IP cores needed as a glue/interface logic between components in the MPSoC. ESPAM converts the application specification at ESL to efficient C/C++ code including code implementing the functional behavior together with code for synchronization of the communication between the processors. This synchronization code contains a memory map of the MPSoC and read/write synchronization primitives. The generated program C/C++ code for each processor in the MPSoC is given to a standard GCC compiler to generate executable code.

In the third and last design phase, a commercial synthesizer converts the generated hardware RTL specification to a GATE-LEVEL specification, thereby generating the target platform gate-level netlist, see the bottom part of Fig. 30.1. This GATE-LEVEL specification is actually the system implementation. In addition, the system implementation is used for validation/calibration of the high-level models in order to improve the accuracy of the design space exploration process at ESL.

Finally, a specific characteristic of the DAEDALUS design flow is that the mapping specification generated by SESAME gives explicitly only the relation between the processes (tasks) in *application specification* and the processing components in *platform specification*. The mapping of FIFO channels to memories is not given explicitly in the mapping specification because, in MPSoCs designed with DAEDALUS, this mapping strictly depends on the mapping of processes to processing components by obeying the following rule. FIFO channel X is always mapped to a local memory of processing component Y if the process that writes to X is mapped on processing component Y. This mapping rule is used by SESAME during the system-level DSE where alternative platform and mapping decisions are explored. The same rule is used by ESPAM (the elaborate mapping step in Fig. 30.7) to explicitly derive the mapping of FIFO channels to memories which is implicit (not explicitly given) in the mapping specification generated by SESAME and forwarded to ESPAM.

30.3 The Polyhedral Process Network Model of Computation for MPSoC Codesign and Programming

In order to facilitate systematic and automated MPSoC codesign and programming, a parallel model of computation (MoC) is required for the application specification at ESL. This is because the MPSoC platforms contain processing components that run in parallel and a parallel MoC represents an application as a composition of concurrent tasks with a well-defined mechanism for inter-task communication and synchronization. Thus, the operational semantics of a parallel MoC match very well the parallel operation of the processing components in an MPSoC. Many parallel MoCs exist [24], and each of them has its own specific characteristics. Evidently, to make the right choice of a parallel MoC, we need to take into account the application domain that is targeted. The DAEDALUS methodology targets streaming (data-flow-dominated) applications in the realm of multimedia, imaging, and signal processing that naturally contain tasks communicating via streams of data. Such applications are very well modeled by using the parallel data-flow MoC called polyhedral process

network (PPN) [30, 31, 54]. Therefore, DAEDALUS uses the PPN model as an application specification at ESL as shown in Fig. 30.1.

A PPN is a network of concurrent processes that communicate through *bounded* first-in first-out (FIFO) channels carrying streams of data tokens. A process produces tokens of data and sends them along a FIFO communication channel where they are stored until a destination process consumes them. FIFO communication channels are the only method which processes may use to exchange data. For each channel there is a single process that produces tokens and a single process that consumes tokens. Multiple producers or multiple consumers connected to the same channel are not allowed. The synchronization between processes is done by *blocking on an empty/full FIFO channel*. Blocking on an empty FIFO channel means that a process is suspended when it attempts to consume data from an empty input channel until there is data in the channel. Blocking on a full FIFO channel means that a process is suspended when it attempts to send data to a full output channel until there is room in the channel. At any given point in time, a process either performs some computation or it is blocked on only one of its channels. A process may access only one channel at a time and when blocked on a channel, a process may not access other channels. An example of a PPN is shown in Fig. 30.2a. It consists of three processes ($P1$, $P2$, and $P3$) that are connected through four FIFO channels ($CH1$, $CH2$, $CH3$, and $CH4$).

The PPN MoC is a special case of the more general Kahn process network (KPN) MoC [20] in the following sense. First, the processes in a PPN are uniformly structured and execute in a particular way. That is, a process first reads data from FIFO channels, then executes some computation on the data, and finally writes results of the computation to FIFO channels. For example, consider the PPN shown

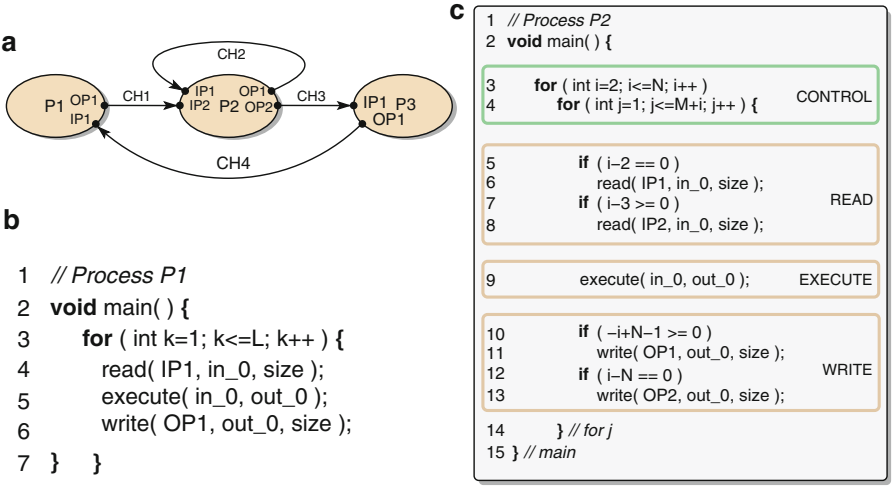


Fig. 30.2 Example of a polyhedral process network and program code of its processes. (**a**) Polyhedral Process Network example. (**b**) Program code for process P1. (**c**) Program code for process P2

in Fig. 30.2a. The program code structure of processes $P1$ and $P2$ are shown in Fig. 30.2b, c, respectively. The structure of the code for both process is the same and consists of a *CONTROL* part, a *READ* part, an *EXECUTE* part, and a *WRITE* part. The difference between $P1$ and $P2$, however, is in the specific code in each part. For example, the *CONTROL* part of $P1$ has only one for loop whereas the *CONTROL* part of $P2$ has two for loops. The blocking synchronization mechanism, explained above, is implemented by read/write synchronization primitives. They are the same for each process. The *READ* part of $P1$ has one read primitive executed unconditionally, whereas the *READ* part of $P2$ has two read primitives and if conditions specifying when to execute these primitives.

Second, the behavior of a process in a PPN can be expressed in terms of parameterized polyhedral descriptions using the polytope model [16], i.e., using formal descriptions of the following form: $D(\mathbf{p}) = \{\mathbf{x} \in \mathbb{Z}^d \mid A \cdot \mathbf{x} \geq B \cdot \mathbf{p} + \mathbf{b}\}$, where $D(\mathbf{p})$ is a parameterized polytope affinely depending on parameter vector $\mathbf{p}$. For example, consider process $P2$ in Fig. 30.2c. The process iterations for which the computational code at line 9 is executed can be expressed as the following two-dimensional polytope: $D_9(N, M) = \{(i, j) \in \mathbb{Z}^2 \mid 2 \leq i \leq N \wedge 1 \leq j \leq M + i\}$. The process iterations for which the read synchronization primitive at line 8 is executed can be expressed as the following two-dimensional polytope: $D_8(N, M) = \{(i, j) \in \mathbb{Z}^2 \mid 3 \leq i \leq N \wedge 1 \leq j \leq M + i\}$. The process iterations for which the other read/write synchronization primitive are executed can be expressed by similar polytopes. All polytopes together capture the behavior of process $P2$, i.e., the code in Fig. 30.2c can be completely constructed from the polytopes and vice versa.

Since PPNs expose task-level parallelism, captured in processes, and make the communication between processes explicit, they are suitable for efficient mapping onto MPSoC platforms. In addition, we motivate our choice of using the PPN MoC in DAEDALUS by observing that the following characteristics of a PPN can take advantage of the parallel resources available in MPSoC platforms:

- **The PPN model is design-time analyzable:** By using the polyhedral descriptions of the processes in a PPN, capacities of the FIFO channels in a PPN, that guarantee deadlock-free execution of the PPN, can be determined at design time;
- **Formal algebraic transformations can be performed on a PPN:** By applying mathematical manipulations on the polyhedral descriptions of the processes in a PPN, the initial PPN can be transformed to an input-output equivalent PPN in order to exploit more efficiently the parallel resources available in an MPSoC platform;
- **The PPN model is determinate:** Irrespective of the schedule chosen to evaluate the network, the same input-output relation always exists. This gives a lot of scheduling freedom that can be exploited when mapping PPNs onto MPSoCs;
- **Distributed Control:** The control is completely distributed to the individual processes and there is no global scheduler present. As a consequence, distributing a PPN for execution on a number of processing components is a relatively simple task;

- **Distributed Memory:** The exchange of data is distributed over FIFO channels. There is no notion of a global memory that has to be accessed by multiple processes. Therefore, resource contention is greatly reduced if MPSoCs with distributed memory are considered;
- **Simple synchronization:** The synchronization between the processes in a PPN is done by a blocking read/write mechanism on FIFO channels. Such synchronization can be realized easily and efficiently in both hardware and software.

Finally, please note that the first and the second bullet, mentioned above, describe characteristics that are specific and valid only for the PPN model. These specific characteristics clearly distinguish the PPN model from the more general KPN model which is used, for example, as an application model in ▸ Chap. 28, "MAPS: A Software Development Environment for Embedded Multicore Applications". The last four bullets above describe characteristics valid for both the PPN and the KPN models.

30.4 Automated Application Parallelization: PNGEN

In this section, we provide an overview of the techniques, we have developed, for automated derivation of PPNs. These techniques are implemented in the PNGEN tool [26, 56] which is part of the DAEDALUS design flow. The input to PNGEN is a SANLP written in *C* and the output is a PPN specification in XML format – see Fig. 30.1. Below, in Sect. 30.4.1, we introduce the SANLPs with their characteristics/limitations and explain how a PPN is derived based on a modified data-flow analysis. We have modified the standard data-flow analysis in order to derive PPNs that have less inter-process FIFO communication channels compared to the PPNs derived by using previous works [23,52]. Then, in Sect. 30.4.2, we explain the techniques to compute the sizes of FIFO channels that guarantee deadlock-free execution of PPNs onto MPSoCs.

30.4.1 SANLPs and Modified Data-Flow Analysis

A SANLP is a sequential program that consists of a set of statements and function calls (the code inside function calls is not limited), where each statement and/or function call is possibly enclosed by one or more loops and/or if statements with the following code limitations: (1) loops must have a constant step size; (2) loops must have bounds that are affine expressions of the enclosing loop iterators, static program parameters, and constants; (3) if statements must have affine conditions in terms of the loop iterators, static program parameters, and constants; (4) the static parameters are symbolic constants, i.e., their values may not change during the execution of the program; (5) the function calls must communicate data between each other explicitly, i.e., using only scalar variables and/or array elements of an arbitrary type that are passed as arguments by value or by reference in function

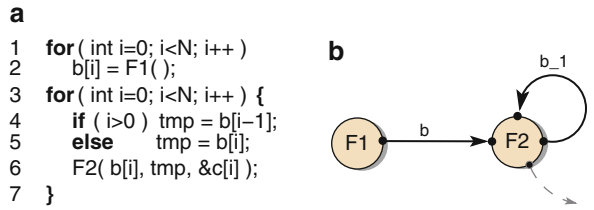


Fig. 30.3 SANLP fragment and its corresponding PPN. (**a**) Example of a SANLP. (**b**) Corresponding PPN

calls; (6) array elements must be indexed with affine expressions of the enclosing loop iterators, static program parameters, and constants. An example of a SANLP that conforms to the abovementioned code limitations is shown in Fig. 30.3a. Other examples can be found in [56]. Although the abovementioned code limitations restrict the expressiveness of a SANLP, in many application domains this is not a problem because it is natural to express an application in the form of a SANLP. Examples are digital signal/image processing and audio/video stream-based applications in consumer electronics, medical imaging, radio astronomy, etc. Some specific application examples are mentioned in Sect. 30.7.

Because of the code limitations, mentioned above, SANLPs can be represented in the well-known polytope model [16], i.e., a compact mathematical representation of a SANLP through sets and relations of integral vectors defined by linear (in)equalities, existential quantification, and the union operation. In particular, the set of iterator vectors for which a function call is executed is an integer set called the *iteration domain*. The linear inequalities of this set correspond to the lower and upper bounds of the loops enclosing the function call. For example, the iteration domain of function `F1` in Fig. 30.3a is $\{i \mid 0 \leq i \leq N-1\}$. Iteration domains form the basis of the description of the processes in the PPN model, as each process corresponds to a particular function call. For example, there are two function calls in the program fragment in Fig. 30.3a representing two application tasks, namely, `F1` and `F2`. Therefore, there are two processes in the corresponding PPN as shown in Fig. 30.3b. The granularity of `F1` and `F2` determines the granularity of the corresponding processes. The FIFO channels are determined by the array (or scalar) accesses in the corresponding function call. All accesses that appear on the left-hand side or in an address of (`&`) expression for an argument of a function call are considered to be *write accesses*. All other accesses are considered to be *read accesses*.

To determine the FIFO channels between the processes, we may perform standard array data-flow analysis [15]. That is, for each execution of a read operation of a given data element in a function call, we need to find the source of the data, i.e., the corresponding write operation that wrote the data element. However, to reduce communication FIFO channels between different processes, in contrast to the standard data-flow analysis and in contrast to [23,52], we also consider all previous read operations from the same function call as possible sources of the data. That is why we call our approach a modified array data-flow analysis [54,56]. The problem to be solved is then: given a read from an array element, what was the last write to

or read from that array element? The last iteration of a function call satisfying some constraints can be obtained by using parametric integer programming (PIP) [14], where we compute the lexicographical *maximum* of the write (or read) source operations in terms of the iterators of the "sink" read operation. Since there may be multiple function calls that are potential sources of the data, and since we also need to express that the source operation is executed before the read (which is not a linear constraint but rather a disjunction of n linear constraints, where n is the shared nesting level), we actually need to perform a number of PIP invocations.

For example, the first read access in function call `F2` of the program fragment in Fig. 30.3a reads data written by function call `F1`, which results in a FIFO channel from process `F1` to process `F2`, i.e., channel `b` in Fig. 30.3b. In particular, data flows from iteration i_w of function `F1` to iteration $i_\mathrm{r} = i_\mathrm{w}$ of function `F2`. This information is captured by the integer relation $D_{\mathtt{F1}\rightarrow\mathtt{F2}} = \{(i_\mathrm{w}, i_\mathrm{r}) \mid i_\mathrm{r} = i_\mathrm{w} \wedge 0 \leq i_\mathrm{r} \leq N-1\}$. For the second read access in function call `F2`, after elimination of the temporary variable `tmp`, the data has already been read by the same function call after it was written. This results in a self-loop channel `b_1` from `F2` to itself described as $D_{\mathtt{F2}\rightarrow\mathtt{F2}} = \{(i_\mathrm{w}, i_\mathrm{r}) \mid i_\mathrm{w} = i_\mathrm{r} - 1 \wedge 1 \leq i_\mathrm{r} \leq N-1\} \cup \{(i_\mathrm{w}, i_\mathrm{r}) \mid i_\mathrm{w} = i_\mathrm{r} = 0\}$. In general, we obtain pairs of write/read and read operations such that some data flows from the write/read operation to the (other) read operation. These pairs correspond to the channels in our process network. For each of these pairs, we further obtain a union of integer relations $\bigcup_{j=1}^{m} D_j(\mathbf{i}_\mathrm{w}, \mathbf{i}_\mathrm{r}) \subset \mathbb{Z}^{n_1} \times \mathbb{Z}^{n_2}$, with n_1 and n_2 the number of loops enclosing the write and read operation, respectively, that connect the specific iterations of the write/read and read operations such that the first is the source of the second. As such, each iteration of a given read operation is uniquely paired off to some write or read operation iteration.

30.4.2 Computing FIFO Channel Sizes

Computing minimal deadlock-free FIFO channel sizes is a nontrivial global optimization problem. This problem becomes easier if we first compute a deadlock-free schedule and then compute the sizes for each channel individually. Note that this schedule is only computed for the purpose of computing the FIFO channel sizes and is discarded afterward because the processes in PPNs are self-scheduled due to the blocking read/write synchronization mechanism. The schedule we compute may not be optimal; however, our computations do ensure that a valid schedule exists for the computed buffer sizes. The schedule is computed using a greedy approach. This approach may not work for process networks in general, but since we consider only static affine nested loop programs (SANLPs), it does work for any PPN derived from a SANLP. The basic idea is to place all iteration domains in a common iteration space at an offset that is computed by the scheduling algorithm. As in the individual iteration spaces, the execution order in this common iteration space is the lexicographical order. By fixing the offsets of the iteration domain in the common space, we have therefore fixed the relative order between any pair of iterations from any pair of iteration domains. The algorithm starts by computing for

any pair of connected processes, the minimal dependence distance vector, a distance vector being the difference between a read operation and the corresponding write operation. Then, the processes are greedily combined, ensuring that all minimal distance vectors are (lexicographically) positive. The end result is a schedule that ensures that every data element is written before it is read. For more information on this algorithm, we refer to [55], where it is applied to perform loop fusion on SANLPs.

After the scheduling, we may consider all FIFO channels to be self-loops of the common iteration space, and we can compute the channel sizes with the following qualification: we will not be able to compute the absolute minimum channel sizes but at best the minimum channel sizes for the computed schedule. To compute the channel sizes, we compute the number of read iterations $R(i)$ that are executed before a given read operation i and subtract the resulting expression from the number of write iterations $W(i)$ that are executed before the given read operation, so the $number\ of\ elements\ in\ FIFO\ at\ operation\ i\ =\ W(i) - R(i)$. This computation can be performed entirely symbolically using the `barvinok` library [57] that efficiently computes the number of integer points in a parametric polytope. The result is a piecewise (quasi-)polynomial in the read iterators and the parameters. The required channel size is the maximum of this expression over all read iterations: $FIFO\ size\ =\ max(\ W(i) - R(i)\ )$. To compute the maximum symbolically, we apply Bernstein expansion [7] to obtain a parametric *upper bound* on the expression.

30.5 Automated System-Level Design Space Exploration: SESAME

In this section, we provide an overview of the methods and techniques we have developed to facilitate automated design space exploration (DSE) for MPSoCs at the electronic system level (ESL). These methods and techniques are implemented in the SESAME tool [8, 11, 39, 42, 53] which is part of the DAEDALUS design flow illustrated in Fig. 30.1. In Sect. 30.5.1, we highlight the basic concept, deployed in SESAME, for system-level DSE of MPSoC platforms. Then, in Sect. 30.5.2, we explain the system-level performance modeling methods and simulation techniques that facilitate the automation of the DSE.

30.5.1 Basic DSE Concept

Nowadays, it is widely recognized that the separation-of-concerns concept [21] is key to achieving efficient system-level design space exploration of complex embedded systems. In this respect, we advocate the use of the popular Y-chart design approach [22] as a basis for (early) system-level design space exploration. This implies that in SESAME, we separate *application models* and *architecture (performance) models* while also recognizing an explicit *mapping step* to map

application tasks onto architecture resources. In this approach, an application model – derived from a specific application domain – describes the functional behavior of an application in a timing and architecture independent manner. A (platform) architecture model – which has been defined with the application domain in mind – defines architecture resources and captures their performance constraints. To perform quantitative performance analysis, application models are first mapped onto and then cosimulated with the architecture model under investigation, after which the performance of each application-architecture combination can be evaluated. Subsequently, the resulting performance numbers may inspire the designer to improve the architecture, restructure/adapt the application(s), or modify the mapping of the application(s). Essential in this approach is that an application model is independent from architectural specifics, assumptions on hardware/software partitioning, and timing characteristics. As a result, application models can be reused in the exploration cycle. For example, a single application model can be used to exercise different hardware/software partitionings and can be mapped onto a range of architecture models, possibly representing different architecture designs.

30.5.2 System-Level Performance Modeling and Simulation

The SESAME system-level modeling and simulation environment [8, 11, 39, 42, 53] facilitates automated performance analysis of MPSoCs according to the Y-chart design approach as discussed in Sect. 30.5.1, recognizing separate application and architecture models. SESAME has also been extended to allow for capturing power consumption behavior and reliability behavior of MPSoC platforms [44, 45, 50].

The layered infrastructure of SESAME's modeling and simulation environment is shown in Fig. 30.4. SESAME maps application models onto architecture models for cosimulation by means of *trace-driven simulation* while using an intermediate mapping layer for scheduling and event-refinement purposes. This trace-driven simulation approach allows for maximum flexibility and model reuse in the process of exploring different MPSoC configurations and mappings of applications to these MPSoC platforms [8, 11]. To actually explore the design space to find good system implementation candidates, SESAME typically deploys a genetic algorithm (GA). For example, to explore different mappings of applications onto the underlying platform architecture, the mapping of application tasks and inter-task communications can be encoded in a chromosome, which is subsequently manipulated by the genetic operators of the GA [9] (see also ▸ Chap. 9, "Scenario-Based Design Space Exploration"). The remainder of this section provides an overview of each of the SESAME layers as shown in Fig. 30.4.

30.5.2.1 Application Modeling

For application modeling within the DAEDALUS design flow, SESAME uses the polyhedral process network (PPN) model of computation, as discussed in Sect. 30.3, in which parallel processes communicate with each other via bounded FIFO

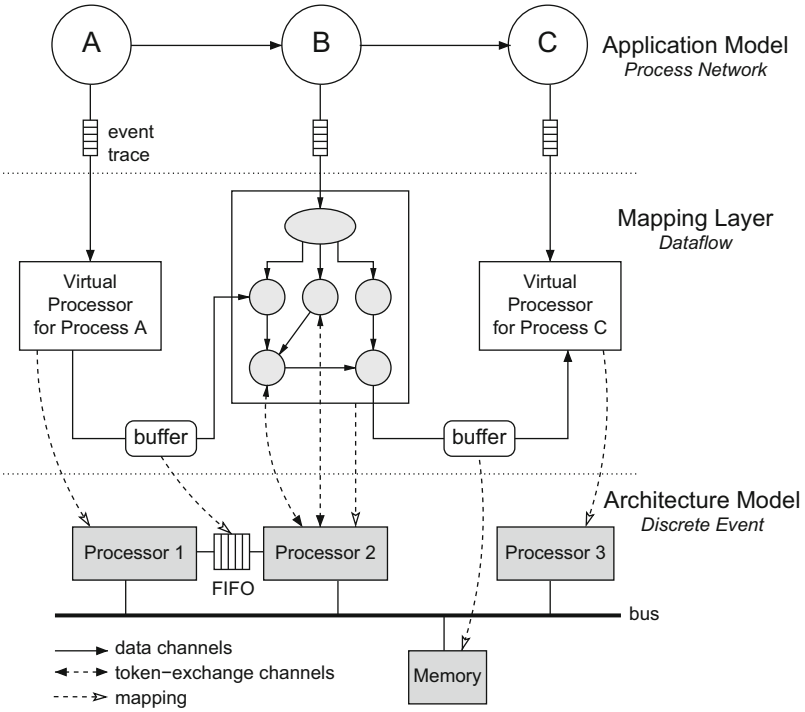


Fig. 30.4 The SESAME's application model layer, architecture model layer, and mapping layer which interfaces between application and architecture models

channels. The PPN application models used in SESAME are either generated by the PNGEN tool presented in Sect. 30.4 or are derived by hand from sequential C/C++ code. The workload of an application is captured by manually instrumenting the code of each PPN process with annotations that describe the application's computational and communication actions, as explained in detail in [8, 11]. By executing the PPN model, these annotations cause the PPN processes to generate traces of *application events* which subsequently drive the underlying architecture model. There are three types of application events: the communication events *read* and *write* and the computational event *execute*. These application events typically are coarse grained, such as *execute(DCT)* or *read(pixel-block,channel_id)*.

To execute PPN application models, and thereby generating the application events that represent the workload imposed on the architecture, SESAME features a process network execution engine supporting the PPN semantics (see Sect. 30.3). This execution engine runs the PPN processes, which are written in C++, as separate threads using the Pthreads package. To allow for rapid creation and modification of models, the structure of the application models (i.e., which processes are used in the model and how they are connected to each other) is not hard-coded in the C++ implementation of the processes. Instead, it is described in a language called

YML (Y-chart modeling language) [8], which is an XML-based language. It also facilitates the creation of libraries of parameterized YML component descriptions that can be instantiated with the appropriate parameters, thereby fostering reuse of (application) component descriptions. To simplify the use of YML even further, a YML editor has also been developed to compose model descriptions using a GUI.

30.5.2.2 Architecture Modeling

The architecture models in SESAME, which typically operate at the so-called transaction level [6, 19], simulate the performance consequences of the computation and communication events generated by an application model. These architecture models solely account for architectural performance constraints and do not need to model functional behavior. This is possible because the functional behavior is already captured in the application models, which subsequently drive the architecture simulation. An architecture model is constructed from generic building blocks provided by a library, see Fig. 30.1, which contains template performance models for processing components (like processors and IP cores), communication components (like busses, crossbar switches, etc.), and various types of memory. The performance parameter values for these models are typically derived from datasheets or from measurements with lower-level simulators or real hardware platforms [43]. The structure of an architecture model – specifying which building blocks are used from the library and the way they are connected – is also described in YML within SESAME.

SESAME's architecture models are implemented using either Pearl [33] or SystemC [19]. Pearl is a small but powerful discrete-event simulation language which provides easy construction of the models and fast simulation [42].

30.5.2.3 Mapping

To map PPN processes (i.e., their event traces) from an application model onto architecture model components, SESAME provides an intermediate *mapping layer*. Besides this mapping function, the mapping layer has two additional functions as will be explained later on: Scheduling of application events when multiple PPN processes are mapped onto a single architecture component (e.g., a programmable processor) and facilitating gradual model refinement by means of trace event refinement.

The mapping layer consists of virtual processor components and FIFO buffers for communication between the virtual processors. There is a one-to-one relationship between the PPN processes in the application model and the virtual processors in the mapping layer. This is also true for the PPN channels and the FIFO buffers in the mapping layer. The only difference is that the buffers in the mapping layer are limited in size, and their size depends on the modeled architecture. As the structure of the mapping layer is equivalent to the structure of the application model under investigation, SESAME provides a tool that is able to automatically generate the mapping layer from the YML description of an application model.

A virtual processor in the mapping layer reads in an application trace from a PPN process via a trace event queue and dispatches the events to a processing

component in the architecture model. The mapping of a virtual processor onto a processing component in the architecture model is freely adjustable (i.e., virtual processors can dispatch trace events to any specified processing component in the architecture model), and this mapping is explicitly described in a YML-based specification. Clearly, this YML mapping description can easily be manipulated by design space exploration engines to, e.g., facilitate efficient mapping exploration. Communication channels, i.e., the buffers in the mapping layer, are also explicitly mapped onto the architecture model. In Fig. 30.4, for example, one buffer is placed in shared memory, while the other buffer is mapped onto a point-to-point FIFO channel between processors 1 and 2.

The mechanism used to dispatch application events from a virtual processor to an architecture model component guarantees deadlock-free scheduling of the application events from different event traces [42]. Please note that, here, we refer to communication deadlocks caused by mapping multiple PPN processes to a single processor and the fact that these processes are not preempted when blocked on, e.g., reading from an empty FIFO buffer (see [42] for a detailed discussion of these deadlock situations). In this event dispatching mechanism, computation events are always directly dispatched by a virtual processor to the architecture component onto which it is mapped. The latter schedules incoming events that originate from different event queues according to a given policy (FCFS, round-robin, or customized) and subsequently models their timing consequences. Communication events, however, are not directly dispatched to the underlying architecture model. Instead, a virtual processor that receives a communication event first consults the appropriate buffer at the mapping layer to check whether or not the communication is safe to take place so that no deadlock can occur. Only if it is found to be safe (i.e., for read events the data should be available and for write events there should be room in the target buffer), then communication events may be dispatched. As long as a communication event cannot be dispatched, the virtual processor blocks. This is possible because the mapping layer executes in the same simulation as the architecture model. Therefore, both the mapping layer and the architecture model share the same simulation-time domain. This also implies that each time a virtual processor dispatches an application event (either computation or communication) to a component in the architecture model, the virtual processor is blocked in simulated time until the event's latency has been simulated by the architecture model. In other words, the individual virtual processors can be seen as abstract representations of application processes at the system architecture level, while the mapping layer can be seen as an abstract OS model.

When architecture model components need to be gradually refined to disclose more implementation details (such as pipelined processing in processor components), this typically implies that the applications events consumed by the architecture model also need to be refined. In SESAME, this is established by an approach in which the virtual processors at the mapping layer are also refined. The latter is done by incorporating data-flow graphs in virtual processors such that it allows us to perform architectural simulation at multiple levels of abstraction without modifying the application model. Fig. 30.4 illustrates this data-flow-based

refinement by refining the virtual processor for process B with a fictive data-flow graph. In this approach, the application event traces specify *what* a virtual processor executes and *with whom* it communicates, while the internal data-flow graph of a virtual processor specifies *how* the computations and communications take place at the architecture level. For more details on how this refinement approach works, we refer the reader to [10,12,41] where the relation between event trace transformations for refinement and data-flow actors at the mapping layer is explained.

30.6 Automated System-Level HW/SW Synthesis and Code Generation: ESPAM

In this section, we present the methods and techniques, we have developed, for systematic and automated system-level HW/SW synthesis and code generation for MPSoC design and programming. These methods and techniques bridge, in a particular way, the *implementation gap* between the electronic system level (ESL) and the register transfer level (RTL) of design abstraction introduced in Sect. 30.1. The methods and techniques are implemented in the ESPAM tool [25, 34, 36, 37] which is part of the DAEDALUS design flow illustrated in Fig. 30.1 and explained in Sect. 30.2. First, in Sect. 30.6.1, we show an example of the ESL input specification for ESPAM that describes an MPSoC. Second, in Sect. 30.6.2, we introduce the system-level platform model used in ESPAM to construct MPSoC platform instances at ESL. Then, in Sect. 30.6.3, we present how an MPSoC platform instance at ESL is refined and translated systematically and automatically to an MPSoC instance at RTL. This is followed by a discussion in Sect. 30.6.4 about the automated programming of the MPSoCs, i.e., the automated code generation done by ESPAM. It includes details on how ESPAM converts processes in a PPN application specification to software code for every programmable processor in an MPSoC. Finally, in Sect. 30.6.5, we present our approach for building heterogeneous MPSoCs where both programmable processors and dedicated IP cores are used as processing components.

30.6.1 ESL Input Specification for ESPAM

Recall from Sect. 30.2 that ESPAM requires as input an ESL specification of an MPSoC that consists of three parts: *platform, application, and mapping specifications*. In this section, we give examples of these three parts (specifications). We will use these examples in our discussion about the system-level HW/SW synthesis and code generation in ESPAM given in Sects. 30.6.3 and 30.6.4.

30.6.1.1 Platform Specification

Consider an MPSoC platform containing four processing components. An example of the ESL platform specification of this MPSoC is depicted in Fig. 30.5a. This specification, in XML format, consists of three parts which define processing components (four processors, lines 2–5), communication component (crossbar, lines

a

```
1  <platform  name =  "myPlatform" >
2    <processor  name = "uP1" > <port name = "IO1" /> </processor>
3    <processor  name = "uP2" > <port name = "IO1" /> </processor>
4    <processor  name = "uP3" > <port name = "IO1" /> </processor>
5    <processor  name = "uP4" > <port name = "IO1" /> </processor>
6
7    <network  name = "CB"   type = "Crossbar" >
8        <port  name = "IO1" />
9        <port  name = "IO2" />
10       <port  name = "IO3" />
11       <port  name = "IO4" />
12   </network>
13
14   <link  name = "BUS1" >
15       <resource  name = "CB"   <port  name = "IO1" />
16       <resource  name = "uP1"  <port  name = "IO1" />
17   </link>
18   <link  name = "BUS2" >
19       <resource  name = "CB"   <port  name = "IO2" />
20       <resource  name = "uP2"  <port  name = "IO1" />
21   </link>
22   <link  name = "BUS3" >
23       <resource  name = "CB"   <port  name = "IO3" />
24       <resource  name = "uP3"  <port  name = "IO1" />
25   </link>
26   <link  name = "BUS4" >
27       <resource  name = "CB"   <port  name = "IO4" />
28       <resource  name = "uP4"  <port  name = "IO1" />
29   </link>
30 </platform>
```

b

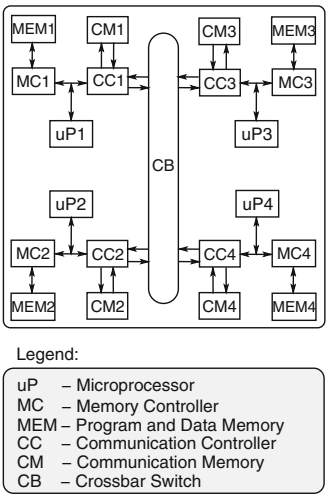


Fig. 30.5 Example of a multiprocessor platform. (**a**) Platform specification. (**b**) Elaborated platform

7–12), and links (lines 14–29). The links specify the connections of the processors to the communication component. Every component has an instance name and different parameters characterizing the component. The components' parameters are not shown in Fig. 30.5a for the sake of brevity. Note that in the specification, there are no memory structures and interface controllers instantiated. The ESPAM tool takes care of this during the platform synthesis described in Sect. 30.6.3. In this way, unnecessary details are hidden at the beginning of the design, keeping the abstraction of the input platform specification very high.

30.6.1.2 Application Specification

Consider an application specified as a PPN consisting of five processes communicating through seven FIFO channels. A graphical representation of the application is shown in Fig. 30.9a. Part of the corresponding XML application specification for this PPN is shown in Fig. 30.6a. Recall that this PPN in XML format is generated automatically by the PNGEN tool using the techniques presented in Sect. 30.4. For the sake of clarity, we show only the description of process $P1$ and channel $FIFO1$ in the XML code. The other processes and channels of the PPN are specified in an identical way. $P1$ has one input port and one output port defined in lines 3–8. $P1$ executes a function called $compute$ (line 9). The function has one input argument (line 10) and one output argument (line 11). There is a strong relation between the function arguments and the ports of a process given at lines 4 and 7. The information how many times function $compute$ has to be fired during the execution of the application is determined by a parameterized *iteration domain* (see

a

```
1  <application name = "myPPN">
2    <process name = "P1">
3      <port name = "p2" direction = "in" >
4        <var name = "in_0" type = "myType" />
5      </port>
6      <port name = "p1" direction = "out" >
7        <var name = "out_0" type = "myType" />
8      </port>
9      <process_code name = "compute" >
10       <arg name = "in_0" type = "input" />
11       <arg name = "out_0" type = "output" />
12       <loop index = "k" parameter = "N" >
13         <loop_bounds matrix = "[1, 1,0,–2; 1,–1,2,–1]"/>
14         <par_bounds matrix = "[1,0,–1,384; 1,0,1,–3]"/>
15       </loop>
16     </process_code>
17   </process>
18       <!–– other processes omitted ––>
19   <channel name = FIFO1 size = "1" >
20     <fromPort name = "p1" />
21     <fromProcess name = "P1"/>
22     <toPort name = "p4" />
23     <toProcess name = "P3"/>
24   </channel>
25       <!–– other channels omitted ––>
26 </application>
```

b

```
1  <mapping name = "myMapping" >
2
3    <processor name = "uP1" >
4      <process name = "P4" />
5    </processor>
6
7    <processor name = "uP2" >
8      <process name = "P2" />
9      <process name = "P5" />
10   </processor>
11
12   <processor name = "uP3" >
13     <proces name = "P3" />
14   </processor>
15
16   <processor name = "uP4" >
17     <process name = "P1" />
18   </processor>
19
20 </mapping>
```

Fig. 30.6 Example of application and mapping specifications. (**a**) Application specification. (**b**) Mapping specification

Sect. 30.4.1) which is captured in a compact (matrix) form at lines 12–15. There are two matrices representing the iteration domain which corresponds to a nested *for*-loop structure. It originates from the structure of the initial (static and affine) nested loop program. In this particular example, there is only one *for* loop with index k and parameter N. The parameter is used in determining the upper bound of the loop. The range of the loop index k is determined at line 13. This matrix represents the following two inequalities, $k - 2 \geq 0$ and $-k + 2N - 1 \geq 0$ and, therefore, $2 \leq k \leq 2N - 1$. In the same way, the matrix at line 14 determines the range of parameter N, i.e., $3 \leq N \leq 384$. Similar information for each port is used to determine at which iterations an input port has to be read and consequently, at which iterations, an output port has to be written. However, for brevity, this information is omitted in Fig. 30.6a. Lines 19–24 show an example of how the topology of a PPN is specified: $FIFO1$ connects processes $P1$ and $P3$ through ports $p1$ and $p4$.

30.6.1.3 Mapping Specification

An example of a mapping specification is shown in Fig. 30.6b. It assumes an MPSoC with four processing components, namely, $uP1$, $uP2$, $uP3$, and $uP4$, and five PPN processes: $P1$, $P2$, $P3$, $P4$, and $P5$. The XML format of the mapping specification is very simple. Process $P4$ is mapped onto processor $uP1$ (see lines 3-5), processes $P2$ and $P5$ are mapped onto processor $uP2$ (lines 7-10), process $P3$ is mapped for execution on processor $uP3$, and, finally, process $P1$ is mapped on processor $uP4$. In the mapping specification, the mapping of FIFO channels to communication memories is not specified. This mapping is related to the way processes are mapped

to processors, and, therefore, the mapping of FIFO channels to communication memories cannot be arbitrary. The mapping of channels is performed by ESPAM automatically which is discussed in Sect. 30.6.3.

30.6.2 System-Level Platform Model

The platform model consists of a library of generic parameterized components and defines the way the components can be assembled. To enable efficient execution of PPNs with low overhead, the platform model allows for building MPSoCs that strictly follow the PPN operational semantics. Moreover, the platform model allows easily to construct platform instances at ESL. To support systematic and automated synthesis of MPSoCs, we have carefully identified a set of components which comprise the MPSoC platforms we consider. It contains the following components.

Processing Components. The processing components implement the functional behavior of an MPSoC. The platform model supports two types of processing components, namely, programmable (ISA) processors and non-programmable, dedicated IP cores. The processing components have several parameters such as *type*, *number of I/O ports*, program and data *memory size*, etc.

Memory Components. Memory components are used to specify the local program and data memories of the programmable processors and to specify data communication storage (buffers) between the processing components (*communication memories*). In addition, the platform model supports dedicated FIFO components used as communication memories in MPSoCs with a point-to-point topology. Important memory component parameters are *type*, *size*, and *number of I/O ports*.

Communication Components. A communication component determines the interconnection topology of an MPSoC platform instance. Some of the parameters of a communication component are *type* and *number of I/O ports*.

Communication Controller. Compliant with our approach to build MPSoCs executing PPNs, communication controllers are used as glue logic realizing the synchronization of the data communication between the processors at hardware level. A communication controller (CC) implements an interface between processing, memory, and communication components. There are two types of CCs in our library. In case of a point-to-point topology, a CC implements only an interface to the dedicated FIFO components used as communication memories. If an MPSoC utilizes a communication component, then the communication controller realizes a multi-FIFO organization of the communication memories. Important CC parameters are *number of FIFOs* and the *size* of each FIFO.

Memory Controllers. Memory controllers are used to connect the local program and data memories to the ISA processors. Every memory controller has a parameter *size* which determines the amount of memory that can be accessed by a processor through the memory controller.

Peripheral Components and **Controllers**. They allow data to be transferred in and out of the MPSoC platform, e.g., a universal asynchronous receive-transmit

(UART). We have also developed a multi-port interface controller allowing for efficient (DMA-like) data communication between the processing cores by sharing an off-chip memory organized as multiple FIFO channels [35]. General off-chip memory controller is also part of this group of library components. In addition, *Timers* can be used for profiling and debugging purposes, e.g., for measuring execution delays of the processing components.

Links. Links are used to connect the components in our system-level platform model. A link is transparent, i.e., it does not have any type, and connects ports of two or more components together.

In DAEDALUS we do not consider the design of processing components. Instead, we use IP cores (programmable processors and dedicated IPs) developed by third parties and propose a communication mechanism that allows efficient data communication (low latency) between these processing components. The devised communication mechanism is independent of the types of processing and communication components used in the platform instance. This results in a platform model that easily can be extended with additional (processing, communication, etc.) components.

30.6.3 Automated System-Level HW Synthesis and Code Generation

The automated translation of an ESL specification of an MPSoC (see Sect. 30.6.1 for an example of such specification) to RTL descriptions goes in three main steps illustrated in Fig. 30.7:

1. **Model initialization.** Using the platform specification, an MPSoC instance is created by initializing an abstract platform model in ESPAM. Based on the application and the mapping specifications, three additional abstract models are initialized: application (ADG), schedule (STree), and mapping models;
2. **System synthesis.** ESPAM elaborates and refines the abstract platform model to a detailed parameterized platform model. Based on the application, schedule, and mapping models, a parameterized process network (PN) model is created as well;
3. **System generation.** Parameters are set and ESPAM generates a platform instance implementation using the RTL version of the components in the library. In addition, ESPAM generates program code for each programmable processor.

30.6.3.1 Model Initialization

In this first step, ESPAM constructs a platform instance from the input platform specification by initializing an abstract platform model. This is done by instantiating and connecting the components in the specification using abstract components from the library. The abstract model represents an MPSoC instance without taking target execution platform details into account. The model includes key system components and their attributes as defined in the platform specification. There are three additional abstract models in ESPAM which are also created and initialized, i.e., an application model, a schedule model, and a mapping model, see the top part

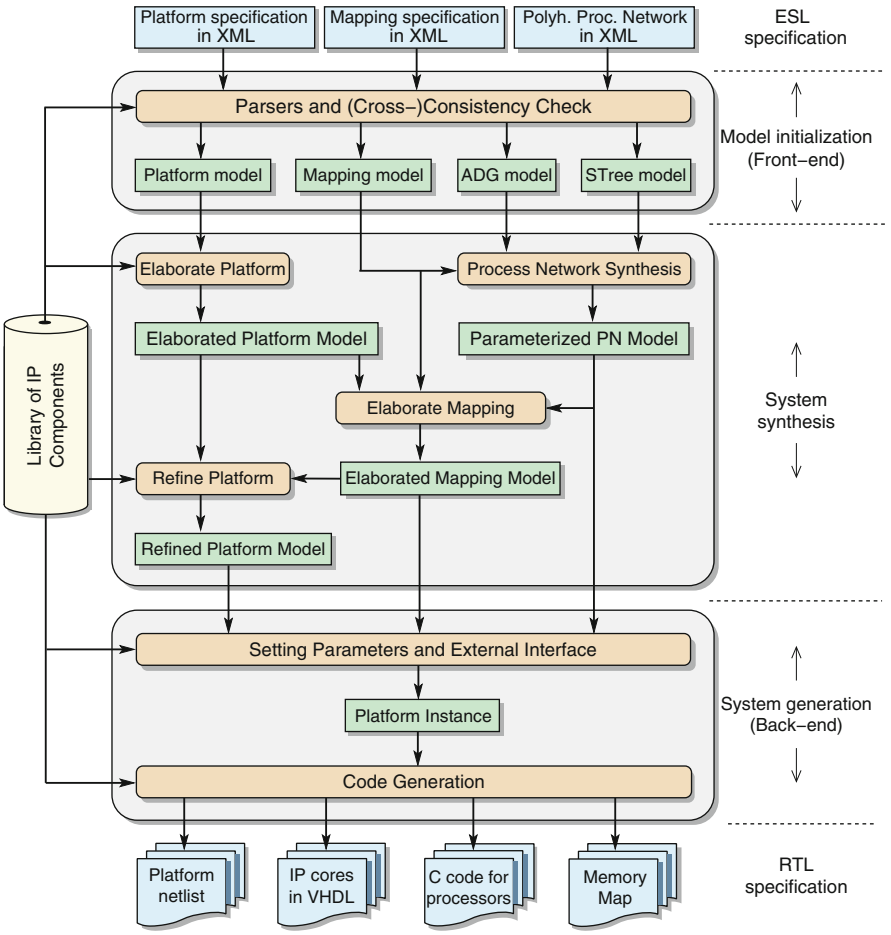


Fig. 30.7 ESL to RTL MPSoC synthesis steps performed by ESPAM

of Fig. 30.7. The application specification consists of two annotated graphs, i.e., a PPN represented by an *approximated dependence graph* (ADG) and a *schedule tree* (STree) representing one valid global schedule of the PPN. Consequently, the ADG and the STree models in ESPAM are initialized, capturing in a formal way all the information that is present in the application specification. Note that, in addition to a standard dependence graph, the ADG is a graph structure that also can capture some data dependencies in an application that are not completely known at design time because the exact application behavior may depend on the data that is processed by the application at run time. If such application is given to ESPAM where some of the data dependencies cannot be exactly determined at design time, then these dependencies are approximated in the ADG. That is, these dependencies are always conservatively put in the ADG, although they may exist only for specific data values processed by the application at run time.

The mapping model is constructed and initialized from the mapping specification. The objective of the mapping model in ESPAM is to capture the relation between the PPN processes in an application and the processing components in an MPSoC instance on the one hand and the relation between FIFO channels and communication memories on the other. The mapping model in ESPAM contains important information which enables the generation of the memory map of the system in an automated way – see Sect. 30.6.4.

30.6.3.2 System Synthesis

The system synthesis step is comprised of several sub-steps. These are platform and mapping model elaboration, process network (PN) synthesis, and platform instance refinement sub-steps. As a result of the platform elaboration, ESPAM creates a detailed parameterized model of a platform instance – see an example of such elaborated platform instance in Fig. 30.5b. The details in this model come from additional components added by ESPAM in order to construct a complete system. In addition, based on the type of the processors instantiated in the first step, the tool automatically synthesizes, instantiates, and connects all necessary communication controllers (CCs) and communication memories (CMs). After the elaboration, a refinement (optimization) step is applied by ESPAM in order to improve resource utilization and efficiency. The refinement step includes program and data memory refinement and compaction in case of processing components with RISC architecture, memory partitioning, and building the communication topology in case of point-to-point MPSoCs. As explained at the end of Sect. 30.2, the mapping specification generated by SESAME contains the relation between PPN processes and processing components only. The mapping of FIFO channels to memories is not given explicitly in the mapping specification. Therefore, ESPAM derives automatically the mapping of FIFO channels to communication memories. This is done in the mapping elaboration step, in which the mapping model is analyzed and augmented with the mapping of FIFO channels to communication memories following the mapping rule described in Sect. 30.2. The PN synthesis is a translation of the *approximated dependence graph* (ADG) model and the *schedule tree* (STree) model into a (parameterized) process network model. This model is used for automated SW synthesis and SW code generation discussed in Sect. 30.6.4.

30.6.3.3 System Generation

This final step consists of a setting parameters sub-step which completely determines a platform instance and a code generation sub-step which generates hardware and software descriptions of an MPSoC. In ESPAM, a software engineering technique called *Visitor* [17] is used to visit the PN and platform model structures and to generate code. For example, ESPAM generates VHDL code for the HW part, i.e., the HW components present in the platform model by instantiating components' templates written in VHDL which are part of the library of IP components. Also, ESPAM generates C/C++ code for the SW part captured in the PN model. The automated SW code generation is discussed in Sect. 30.6.4. The HW description generated by ESPAM consists of two parts: **(1) Platform topology.** This is a netlist

description defining the MPSoC topology that corresponds to the platform instance synthesized by ESPAM. This description contains the components of the platform instance with the appropriate values of their parameters and the connections between the components in the form compliant with the input requirements of the commercial tool used for low-level synthesis. **(2) Hardware descriptions of the MPSoC components.** To every component in the platform instance corresponds a detailed description at RTL. Some of the descriptions are predefined (e.g., processors, memories, etc.), and ESPAM selects them from the library of components and sets their parameters in the platform netlist. However, some descriptions are generated by ESPAM, e.g., an IP Module used for integrating a third-party IP core as a processing component in an MPSoC (discussed in Sect. 30.6.5).

30.6.4 Automated System-Level SW Synthesis and Code Generation

In this section, we present in detail our approach for systematic and automated programming of MPSoCs synthesized with ESPAM. For the sake of clarity, we explain the main steps in the ESPAM programming approach by going through an illustrative example considering the input platform, application, and mapping specifications described in Sect. 30.6.1. For these example specifications, we show how the SW code for each processor in an MPSoC platform is generated and present our SW synchronization and communication primitives inserted in the code. Finally, we explain how the memory map of the MPSoC is generated.

30.6.4.1 SW Code Generation for Processors

ESPAM uses the initial sequential application program, the corresponding PPN application specification, and the mapping specification to generate automatically software (C/C++) code for each processor in the platform specification. The code for a processor contains $control$ code and $computation$ code. The $computation$ code transforms the data that has to be processed by a processor, and it is grouped into function calls in the initial sequential program. ESPAM extracts this code directly from the sequential program. The $control$ code (for loops, if statements, etc.) determines the control flow, i.e., when and how many times data reading and data writing have to be performed by a processor as well as when and how many times the $computation$ code has to be executed in a processor. The $control$ code of a processor is generated by ESPAM according to the PPN application specification and the mapping specification as we explain below.

According to the mapping specification in Fig. 30.6b, process $P1$ is mapped onto processor $uP4$ (see lines 16–18). Therefore, ESPAM uses the XML specification of process $P1$ shown in Fig. 30.6a to generate the $control\ C$ code for processor $uP4$. The code is depicted in Fig. 30.8a. At lines 4–7, the type of the data transferred through the FIFO channels is declared. The data type can be a scalar or more complex type. In this example, it is a structure of 1 Boolean variable and a 64-element array of integers, a data type found in the initial sequential program. There is one parameter (N) that has to be declared as well. This is done at line 8

a

```
1 #include "primitives.h"
2 #include "memoryMap.h"
3
4 struct myType {
5     bool flag;
6     int  data[64];
7 };
8 int N = 384;
9
10 void main( ) {
11     myType in_0;
12     myType out_0;
13
14     for ( int k=2; k<=2*N-1; k++ ) {
15         read( p2, &in_0, sizeof(myType) );
16         compute( in_0, &out_0 );
17         write( p1, &out_0, sizeof(myType) );
18     }
19 }
```

b

```
1 void read( int* port, void* data, int length ) {
2     int *req_&_rd = 0xE0000000;  // Address in a CC
3     int *isEmpty = req_&_rd + 1;
4     *req_&_rd = 0x80000000 | (port);   // Write a request
5     for ( int i=0; i<length; i++ ) {
6         // reading is blocked if a FIFO is empty
7         while ( *isEmpty ) { }
8         (byte* data)[i] = *req_&_rd; // read from a FIFO
9     }
10    *req_&_rd = 0x7FFFFFFF&(inPort);
11 }
12
13 void write( int* port, void* data, int length ) {
14     int *isFull = port + 1;
15     for ( int i=0; i<length; i++ ) {
16         // writing is blocked if a FIFO is full
17         while ( *isFull ) { }
18         *port = (byte* data)[i]; // write to a FIFO
19     }
20 }
```

Fig. 30.8 Source code generated by ESPAM. (**a**) Control code for processor $uP4$. (**b**) Read and write communication primitives

in Fig. 30.8a. Then, at lines 10–19 in the same figure, the behavior of processor $uP4$ is described. In accordance with the XML specification of process $P1$ in Fig. 30.6a, the function $compute$ is executed $2 * N - 2$ times. Therefore, a *for* loop is generated in the $main$ routine for processor $uP4$ in lines 14-18 in Fig. 30.8a. The $computation$ code in function $compute$ is extracted from the initial sequential program. This code is not important for our example; hence, it is not given here for the sake of brevity. The function $compute$ uses local variables in_0 and out_0 declared in lines 11 and 12 in Fig. 30.8a. The input data comes from $FIFO2$ through port $p2$, and the results are written to $FIFO1$ through port $p1$ – see Fig. 30.9a. Therefore, before the function call, ESPAM inserts a *read* primitive to read from $FIFO2$ initializing variable in_0 and after the function call, ESPAM inserts a *write* primitive to send the results (the value of variable out_0) to $FIFO1$ as shown in Fig. 30.8a at lines 15 and 17. When several processes are mapped onto one processor, a schedule is required in order to guarantee a proper execution order of these processes onto one processor. The ESPAM tool automatically finds a local static schedule from the STree model (see Sect. 30.6.3) based on the grouping technique for processes presented in [48].

30.6.4.2 SW Communication and Synchronization Primitives

Recall from Sect. 30.6.2 that the FIFO channels are mapped onto the communication memories of an MPSoC platform instances and the multi-FIFO organization of a communication memory is realized by the corresponding communication controller (CC). A FIFO channel is seen by a processor as two memory locations in its communication memory address space. A processor uses the first location to read the status of the FIFO. The status indicates whether a FIFO is full (data cannot be written) or empty (data is not available). This information is used for the

inter-processor synchronization. The second location is used to read/write data from/to the FIFO buffer, thereby, realizing inter-processor data transfer.

The described behavior is realized by the SW communication and synchronization primitives interacting with the HW communication controllers. The code implementing the *read* and *write* primitives used in lines 15 and 17 in Fig. 30.8a is shown in Fig. 30.8b. Both read and write primitives have three parameters: $port$, $data$, and $length$. Parameter $port$ is the address of the memory location through which a processor can access a given FIFO channel for reading/writing. Parameter $data$ is a pointer to a local variable and $length$ specifies the amount of data (in bytes) to be moved from/to the local variable to/from the FIFO channel. The primitives implement the blocking synchronization mechanism between the processors in the following way. First, the status of a channel that has to be read/written is checked. A channel status is accessed using the locations defined in lines 3 and 14. The blocking is implemented by while loops with empty bodies (busy-polling mechanism) in lines 7 and 17. A loop iterates (does nothing) while a channel is full or empty. Then, in lines 8 and 18 the actual data transfer is performed. Note that the busy-polling mechanism, described above, to implement the blocking is sufficient because PPN processes mapped onto a processor are statically scheduled, and the busy-polling mechanism exactly follows/implements the blocking semantics of a PPN process, discussed in the second paragraph of Sect. 30.3, thereby guaranteeing deterministic execution of the PPN.

30.6.4.3 Memory Map Generation

Each FIFO channel in an MPSoCs has separate read and write ports. A processor accesses a FIFO for read operations using the *read* synchronization primitive. The parameter $port$ specifies the address of the read port of the FIFO channel to be accessed. In the same way, the processor writes to a FIFO using the write synchronization primitive where the parameter $port$ specifies the address of the write port of this FIFO. The FIFO channels are implemented in the communication memories (CMs); therefore, the addresses of the FIFO ports are located in the processors' address space where the communication memory segment is defined. The memory map of an MPSoC generated by ESPAM contains the values defining the read and the write addresses of each FIFO channel in the system.

The first step in the memory map generation is the mapping of the FIFO channels in the PPN application specification onto the communication memories (CMs) in the multiprocessor platform. This mapping cannot be arbitrary and should obey the mapping rule described at the end of Sect. 30.2. That is, ESPAM maps FIFO channels onto CMs of processors in the following automated way. First, for each process in the application specification ESPAM finds all the channels this process writes to. Then, from the mapping specification ESPAM finds which processor corresponds to the current process and maps the found channels in the processor's local CM. For example, consider the mapping specification shown in Fig. 30.6b which defines only the mapping of the processes of the PPN in Fig. 30.9a to the processors in the platform shown in Fig. 30.9b. Based on this mapping specification, ESPAM maps automatically $FIFO2$, $FIFO3$, and $FIFO5$ onto the

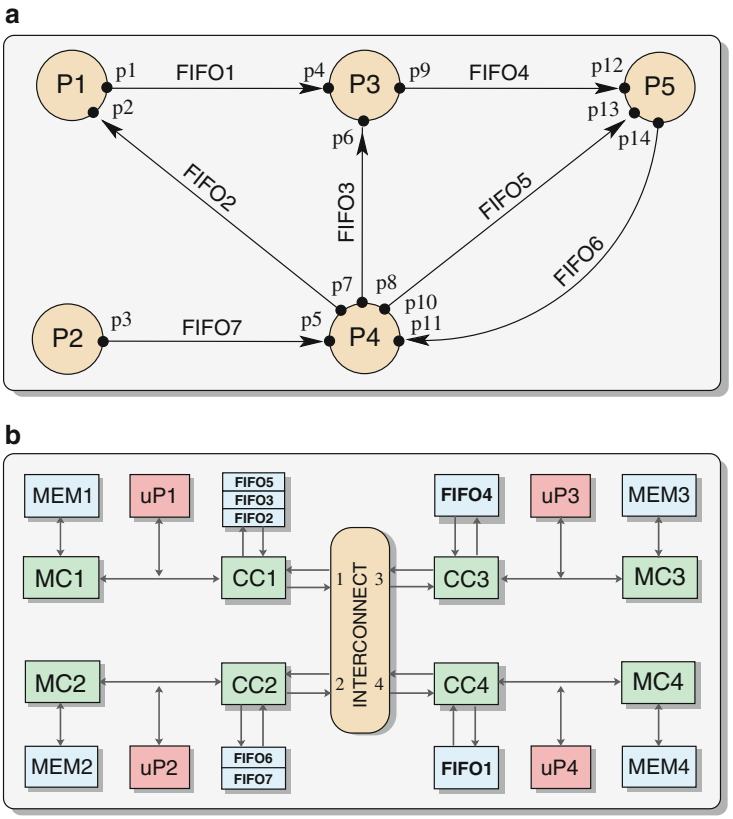


Fig. 30.9 Mapping example. (**a**) Polyhedral process network. (**b**) Example platform

CM of processor $uP1$ because process $P4$ is mapped onto processor $uP1$ and process $P4$ writes to channels $FIFO2$, $FIFO3$, and $FIFO5$. Similarly, $FIFO4$ is mapped onto the CM of processor $uP3$, and $FIFO1$ is mapped onto the CM of $uP4$. Since both processes $P2$ and $P5$ are mapped onto processor $uP2$, ESPAM maps $FIFO6$ and $FIFO7$ onto the CM of this processor.

After the mapping of the channels onto the CMs, ESPAM generates the memory map of the MPSoC, i.e., generates values for the FIFOs' read and write addresses. For the mapping example illustrated in Fig. 30.9b, the generated memory map is shown in Fig. 30.10. Notice that $FIFO1$, $FIFO2$, $FIFO4$, and $FIFO6$ have equal write addresses (see lines 4, 6, 10, and 14). This is not a problem because writing to these FIFOs is done by different processors, and these FIFOs are located in the local CMs of these different processors, i.e., these addresses are local processor write addresses. The same applies for the write addresses of $FIFO3$ and $FIFO7$. However, all processors can read from all FIFOs via a communication component. Therefore, the read addresses have to be unique in the

```
1  #ifndef _MEMORYMAP_H_
2  #define _MEMORYMAP_H_
3
4  #define p1   0xe0000008 //write addr. FIFO1
5  #define p4   0x00040001 //read addr. FIFO1
6  #define p7   0xe0000008 //write addr. FIFO2
7  #define p2   0x00010001 //read addr. FIFO2
8  #define p8   0xe0000010 //write addr. FIFO3
9  #define p6   0x00010002 //read addr. FIFO3
10 #define p9   0xe0000008 //write addr. FIFO4
11 #define p12  0x00030001 //read addr. FIFO4
12 #define p10  0xe0000018 //write addr. FIFO5
13 #define p13  0x00010003 //read addr. FIFO5
14 #define p14  0xe0000008 //write addr. FIFO6
15 #define p11  0x00020001 //read addr. FIFO6
16 #define p3   0xe0000010 //write addr. FIFO7
17 #define p5   0x00020002 //read addr. FIFO7
18
19 #endif
```

Fig. 30.10 The memory map of the MPSoC platform instance generated by ESPAM

MPSoC memory map and the read addresses have to specify precisely the CM in which a FIFO is located. To accomplish this, a read address of a FIFO has two fields: a communication memory (CM) number and a FIFO number within a CM.

Consider, for example, $FIFO3$ in Fig. 30.9b. It is the second FIFO in the CM of processor $uP1$; thus this FIFO is numbered with 0002 in this CM. Also, the CM of $uP1$ can be accessed for reading through port 1 of the communication component *INTERCONNECT* as shown in Fig. 30.9b; thus this CM is uniquely numbered with 0001. As a consequence, the unique read address of $FIFO3$ is determined to be 0x00010002 – see line 9 in Fig. 30.10, where the first field 0001 is the CM number and the second field 0002 is the FIFO number in this CM. In the same way, ESPAM determines automatically the unique read addresses of the rest of the FIFOs that are listed in Fig. 30.10.

30.6.5 Dedicated IP Core Integration with ESPAM

In Sects. 30.6.3 and 30.6.4 we presented our approach to system-level HW/SW synthesis and code generation for MPSoCs that contain only programmable (ISA) processing components. Based on that, in this section, we present an overview of our approach to augment these MPSoCs with non-programmable dedicated IP cores in a systematic and automated way. Such an approach is needed because, in some cases, an MPSoC that contains only programmable processors may not meet the performance requirements of an application. For better performance and efficiency, in a multiprocessor system, some application tasks may have to be executed by

dedicated (customized and optimized) IP cores. Moreover, many companies already provide dedicated customizable IP cores optimized for a particular functionality that aim at saving design time and increasing overall system performance and efficiency. Therefore, we have developed techniques, implemented in ESPAM, for automated generation of an *IP Module* which is a wrapper around a dedicated and predefined IP core. The generated IP Module allows ESPAM to integrate an IP core into an MPSoC in an automated way. The generation of IP Modules is based on the properties of the PPN application model we use in DAEDALUS. Below, we present the basic idea in our IP integration approach. It is followed by a discussion on the type of the IPs supported by ESPAM and the interfaces these IPs have to provide in order to allow automated integration. More details about our integration approach can be found in [36].

30.6.5.1 IP Module: Basic Idea and Structure

As we explained earlier, in the multiprocessor platforms we consider, the processors execute code implementing PPN processes and communicate data between each other through FIFO channels mapped onto communication memories. Using communication controllers, the processors can be connected via a communication component. We follow a similar approach to connect an IP Module to other IP Modules or programmable processors in an MPSoCs. We illustrate our approach with the example depicted in Fig. 30.11. We map the PPN in Fig. 30.2a onto the heterogeneous platform shown in Fig. 30.11a. Assume that process $P1$ is executed by processor $uP1$, $P3$ is executed by $uP2$, and the functionality of process $P2$ is implemented as a dedicated (predefined) IP core embedded in an IP Module. Based on this mapping and the PPN topology, ESPAM automatically maps FIFO channels to communication memories (CMs) following the rule that a processing component

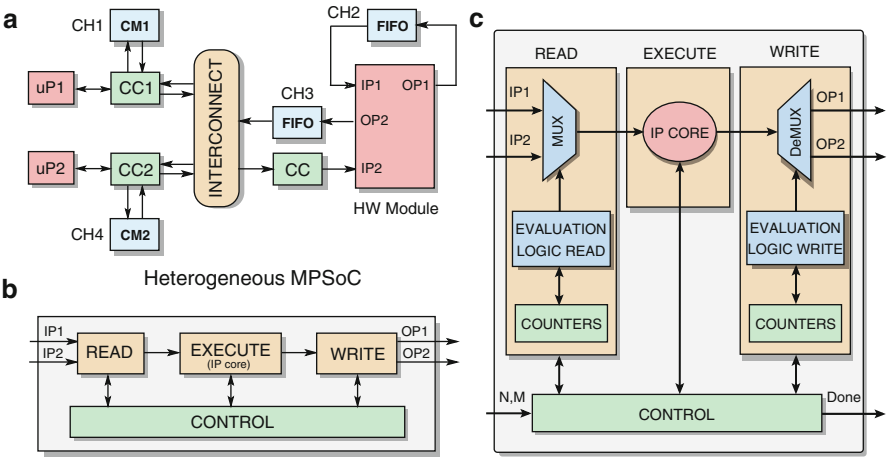


Fig. 30.11 Example of heterogeneous MPSoC generated by ESPAM. (**a**) Heterogeneous MPSoC. (**b**) Top-level view of the IP module. (**c**) IP module structure

only writes to its local CM. For example, process $P1$ is mapped onto processing component $uP1$ and $P1$ writes to FIFO channel $CH1$. Therefore, $CH1$ is mapped onto the local CM of $uP1$ – see Fig. 30.11a. In order to connect a dedicated IP core to other processing components, ESPAM generates an IP Module (IPM) that contains the IP core and a wrapper around it. Such an IPM is then connected to the system using communication controllers (CCs) and communication memories (CMs), i.e., an IPM writes directly to its own local FIFOs and uses CCs (one CC for every input of an IP core) to read data from FIFOs located in CMs of other processors. The IPM that realizes process $P2$ is shown in Fig. 30.11b.

As explained in Sect. 30.3, the processes in a PPN have always the same structure. It reflects the PPN operational semantics, i.e, read-execute-write using blocking read/write synchronization mechanism. Therefore, an IP Module realizing a process of a PPN has the same structure, shown in Fig. 30.11b, consisting of *READ*, *EXECUTE*, and *WRITE* components. A *CONTROL* component is added to capture the process behavior, e.g., the number of process firings, and to synchronize the operation of components *READ*, *EXECUTE*, and *WRITE*. The *EXECUTE* component of an IPM is actually the dedicated IP core to be integrated. It is not generated by ESPAM but it is taken from a library. The other components *READ*, *WRITE*, and *CONTROL* constitute the wrapper around the IP core. The wrapper is generated fully automatically by ESPAM based on the specification of a process to be implemented by the given IPM. Each of the components in an IPM has a particular structure which we illustrate with the example in Fig. 30.11c. Figure 30.2c shows the specification of process $P2$ in the PPN of Fig. 30.2a if $P2$ would be executed on a programmable processor. We use this code to show the relation with the structure of each component in the IP Modules generated by ESPAM, shown in Fig. 30.11c, when $P2$ is realized by an IP Module.

In Fig. 30.2c, the read part of the code is responsible for getting data from proper FIFO channels at each firing of process $P2$. This is done by the code lines 5–8 which behave like a multiplexer, i.e., the internal variable in_0 is initialized with data taken either from port $IP1$ or $IP2$. Therefore, the read part of $P2$ corresponds to the multiplexer MUX in the *READ* component of the IP Module in Fig. 30.11c. Selecting the proper channel at each firing is determined by the *if* conditions at lines 5 and 7. These conditions are realized by the EVALUATION LOGIC READ sub-component in component *READ*. The output of this sub-component controls the MUX sub-component. To evaluate the *if* conditions at each firing, the iterators of the *for* loops at lines 3 and 4 are used. Therefore, these *for* loops are implemented by counters in the IP Module – see the COUNTERS sub-component in Fig. 30.11c.

The write part in Fig. 30.2c is similar to the read part. The only difference is that the write part is responsible for writing the result to proper channels at each firing of $P2$. This is done in code lines 10–13. This behavior is implemented by the demultiplexer DeMUX sub-component in the *WRITE* component in Fig. 30.11c. DeMUX is controlled by the EVALUATION LOGIC WRITE sub-component which implements the *if* conditions at lines 10 and 12. Again, to implement the *for* loops, ESPAM uses a COUNTERS sub-component. Although, the counters correspond to the control part of process $P2$, ESPAM implements them in both the *READ* and

WRITE blocks, i.e., it duplicates the *for*-loops implementation in the IP Module. This allows the operation of components *READ*, *EXECUTE*, and *WRITE* to overlap, i.e., they can operate in pipeline which leads to better performance of the IP Module.

The execute part in Fig. 30.2c represents the main computation in $P2$ encapsulated in the function call at code line 9. The behavior inside the function call is realized by the dedicated IP core depicted in Fig. 30.11c. As explained above, this IP core is not generated by ESPAM but it is taken from a library of predefined IP cores provided by a designer. An IP core can be created by hand or it can be generated automatically from *C* descriptions using high-level synthesis tools like, e.g., Xilinx Vivado [58]. In the IP Module, the output of sub-component MUX is connected to the input of the IP core, and the output of the IP core is connected to the input of sub-component DeMUX. In the example, the IP core has one input and one output. In general, the number of inputs/outputs can be arbitrary. Therefore, every IP core input is connected to one MUX and every IP core output is connected to one DeMUX.

Notice that the loop bounds at lines 3–4 in Fig. 30.2c are parameterized. The *CONTROL* component in Fig. 30.11c allows the parameter values to be set/modified from outside the IP Module at run time or to be fixed at design time. Another function of component *CONTROL* is to synchronize the operation of the IP Module components and to make them to work in pipeline. Also, *CONTROL* implements the blocking read/write synchronization mechanism. Finally, it generates the status of the IP Module, i.e., signal $Done$ indicates that the IP Module has finished an execution.

30.6.5.2 IP Core Types and Interfaces

In this section we describe the type of the IP cores that fit in our IP Module idea and structure discussed above. Also, we define the minimum data and control interfaces these IP cores have to provide in order to allow automated integration in MPSoC platforms generated by ESPAM.

1. In the IP Module, an IP core implements the main computation of a PPN process which in the initial sequential application specification is represented by a function call. Therefore, an IP core has to behave like a function call as well. This means that for each input data, read by the IP Module, the IP core is *executed* and produces output data after an arbitrary delay;
2. In order to guarantee seamless integration within the data-flow of our heterogeneous systems, an IP core must have unidirectional data interfaces at the input and the output that do not require random access to read and write data from/to memory. Good examples of such IP cores are the *multimedia cores* at http://www.cast-inc.com/cores/;
3. To synchronize an IP core with the other components in the IP Module, the IP core has to provide `Enable/Valid` control interface signals. The `Enable` signal is a control input to the IP core and is driven by the *CONTROL* component in the IP Module to enable the operation of the IP core when input data is read

from input FIFO channels. If input data is not available, or there is no room to store the output of the IP core to output FIFO channels, then `Enable` is used to suspend the operation of the IP core. The `Valid` signal is a control output signal from the IP and is monitored by component *CONTROL* in order to ensure that only valid data is written to output FIFO channels connected to the IP Module.

30.7 Summary of Experiments and Results

As a proof of concept, the DAEDALUS methodology/framework and its individual tools (PNGEN, SESAME, and ESPAM) have been tested and evaluated in experiments and case studies considering several streaming applications with different complexity ranging from image processing kernels, e.g., Sobel filter and discrete wavelet transform (DWT), to complete applications, e.g., Motion-JPEG encoder (MJPEG), JPEG2000 codec, JPEG encoder, H.264 decoder, and medical image registration (MIR). For the description of these experiments, case studies, and the obtained results, we refer the reader to the following publications: [36,37] for Sobel and DWT, [34, 36, 37, 51] for MJPEG, [1] for JPEG2000, [38] for JPEG, [46] for H.264, and [13] for MIR. In this section, we summarize very briefly the JPEG encoder case study [38] in order to highlight the improvements, in terms of performance and design productivity, that can be achieved by using DAEDALUS on an industry-relevant application. This case study, which we conducted in a project together with an industrial partner, involves the design of a JPEG-based image compression MPSoC for very high-resolution (in the order of gigapixels) cameras targeting medical appliances. In this project, the DAEDALUS framework was used for design space exploration (DSE) and MPSoC implementation, both at the level of simulations and real MPSoC prototypes, in order to rapidly gain detailed insight on the system performance. Our experience showed that all conducted DSE experiments and the real implementation of 25 MPSoCs (13 of them were heterogeneous MPSoCs) on an FPGA were performed in a short amount of time, 5 days in total, due to the highly automated DAEDALUS design flow. Around 70% of this time was taken by the low-level commercial synthesis and place-and-route FPGA tools. The obtained implementation results showed that the DAEDALUS high-level MPSoC models were capable of accurately predicting the overall system performance, i.e., the performance error was around 5%. By exploiting the data- and task-level parallelism in the JPEG application, DAEDALUS was able to deliver scalable MPSoC solutions in terms of performance and resource utilization. We were able to achieve a performance speedup of up to 20x compared to a single processor system. For example, a performance speedup of 19.7x was achieved on a heterogeneous MPSoC which utilizes 24 parallel cores, i.e., 16 MicroBlaze programmable processor cores and 8 dedicated hardware IP cores. The dedicated hardware IP cores implement the Discrete Cosine Transform (DCT) within the JPEG application. The MPSoC system performance was limited by the available on-chip FPGA memory resources and the available dedicated hardware IP cores in the DAEDALUS RTL library (we had only the dedicated DCT IP core available).

30.8 Conclusions

In this chapter, we have presented our system design methods and techniques that are implemented and integrated in the DAEDALUS design/tool flow for automated system-level synthesis, implementation, and programming of streaming multiprocessor embedded systems on chips. DAEDALUS features automated application parallelization (the PNGEN tool), automated system-level DSE (the SESAME tool), and automated system-level HW/SW synthesis and code generation (the ESPAM tool). This automation significantly reduces the design time starting from a functional specification and going down to complete MPSoC implementation. Many experiments and case studies have been conducted using DAEDALUS, and we could conclude that DAEDALUS helps an MPSoC designer to reduce the design and programming time from several months to only a few days as well as to obtain high quality MPSoCs in terms of performance and resource utilization.

In addition to the well-established methods and techniques, presented in this chapter, DAEDALUS has been extended with new advanced techniques and tools for designing *hard-real-time* embedded streaming MPSoCs. This extended version of DAEDALUS is called DAEDALUS^{RT} [2–5,28,47]. Its extra features are (1) support for multiple applications running simultaneously on an MPSoC; (2) very fast, yet accurate, schedulability analysis to determine the minimum number of processors needed to schedule the applications; and (3) usage of hard-real-time multiprocessor scheduling algorithms providing temporal isolation to schedule the applications.

References

1. Azkarate-askasua M, Stefanov T (2008) JPEG2000 image compression in multi-processor system-on-chip. Tech. rep., CE-TR-2008-05, Delft University of Technology, The Netherlands
2. Bamakhrama M, Stefanov T (2011) Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In: Proceedings of the EMSOFT 2011, pp 195–204
3. Bamakhrama M, Stefanov T (2012) Managing latency in embedded streaming applications under hard-real-time scheduling. In: Proceedings of the CODES+ISSS 2012, pp 83–92
4. Bamakhrama M, Stefanov T (2013) On the hard-real-time scheduling of embedded streaming applications. Des Autom Embed Syst 17(2):221–249
5. Bamakhrama M, Zhai J, Nikolov H, Stefanov T (2012) A methodology for automated design of hard-real-time embedded streaming systems. In: Proceedings of the DATE 2012, pp 941–946
6. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of the CODES+ISSS 2003, pp 19–24
7. Clauss P, Fernandez F, Garbervetsky D, Verdoolaege S (2009) Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. IEEE Trans VLSI Syst 17(8):983–996
8. Coffland JE, Pimentel AD (2003) A software framework for efficient system-level performance evaluation of embedded systems. In: Proceedings of the SAC 2003, pp 666–671
9. Erbas C, Cerav-Erbas S, Pimentel AD (2006) Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. IEEE Trans Evol Comput 10(3):358–374
10. Erbas C, Pimentel AD (2003) Utilizing synthesis methods in accurate system-level exploration of heterogeneous embedded systems. In: Proceedings of the SiPS 2003, pp 310–315

11. Erbas C, Pimentel AD, Thompson M, Polstra S (2007) A framework for system-level modeling and simulation of embedded systems architectures. EURASIP J Embed Syst 2007(1):1–11
12. Erbas C, Polstra S, Pimentel AD (2003) IDF models for trace transformations: a case study in computational refinement. In: Proceedings of the SAMOS 2003, pp 178–187
13. Farago T, Nikolov H, Klein S, Reiber J, Staring M (2010) Semi-automatic parallelisation for iterative image registration with B-splines. In: International workshop on high-performance medical image computing for image-assisted clinical intervention and decision-making (HP-MICCAI'10)
14. Feautrier P (1988) Parametric integer programming. Oper Res 22(3):243-268
15. Feautrier P (1991) Dataflow analysis of scalar and array references. Int J Parallel Program 20(1):23–53
16. Feautrier P (1996) Automatic parallelization in the polytope model. In: Perrin GR, Darte A (eds) The data parallel programming model. Lecture notes in computer science, vol 1132. Springer, Berlin/Heidelberg, pp 79–103
17. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston
18. Gerstlauer A, Haubelt C, Pimentel A, Stefanov T, Gajski D, Teich J (2009) Electronic System-level synthesis methodologies. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(10):1517–1530
19. Grötker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic, Dordrecht
20. Kahn G (1974) The semantics of a simple language for parallel programming. In: Proceedings of the IFIP Congress 74. North-Holland Publishing Co.
21. Keutzer K, Newton A, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543
22. Kienhuis B, Deprettere EF, van der Wolf P, Vissers KA (2002) A methodology to design programmable embedded systems: the Y-chart approach. In: Embedded processor design challenges, LNCS, vol 2268. Springer, pp 18–37
23. Kienhuis B, Rijpkema E, Deprettere E (2000) Compaan: deriving process networks from Matlab for embedded signal processing architectures. In: Proceedings of the CODES 2000, pp 13–17
24. Lee E, Sangiovanni-Vincentelli A (1998) A framework for comparing models of computation. IEEE Trans Comput-Aided Des Integr Circuits Syst 17(12):1217–1229
25. Leiden University: The ESPAM tool. http://daedalus.liacs.nl/espam/
26. Leiden University: The PNgen tool. http://daedalus.liacs.nl/pngen/
27. Leiden University and University of Amsterdam: The DAEDALUS System-level Design Framework. http://daedalus.liacs.nl/
28. Liu D, Spasic J, Zhai J, Stefanov T, Chen G (2014) Resource optimization for CSDF-modeled streaming applications with latency constraints. In: Proceedings of the DATE 2014, pp 1–6
29. Martin G (2006) Overview of the MPSoC design challenge. In: Proceedings of the design automation conference (DAC'06), pp 274–279
30. Meijer S, Nikolov H, Stefanov T (2010) Combining process splitting and merging transformations for polyhedral process networks. In: Proceedings of the ESTIMedia 2010, pp 97–106
31. Meijer S, Nikolov H, Stefanov T (2010) Throughput modeling to evaluate process merging transformations in polyhedral process networks. In: Proceedings of the DATE 2010, pp 747–752
32. Mihal A, Keutzer K (2003) Mapping concurrent applications onto architectural platforms. In: Jantsch A, Tenhunen H (eds) Networks on chip. Kluwer Academic Publishers, Boston, pp 39–59
33. Muller HL (1993) Simulating computer architectures. Ph.D. thesis, Department of Computer Science, University of Amsterdam
34. Nikolov H, Stefanov T, Deprettere E (2006) Multi-processor system design with ESPAM. In: Proceedings of the CODES+ISSS 2006, pp 211–216

35. Nikolov H, Stefanov T, Deprettere E (2007) Efficient external memory interface for multi-processor platforms realized on FPGA chips. In: Proceedings of the FPL 2007, pp 580–584
36. Nikolov H, Stefanov T, Deprettere E (2008) Automated integration of dedicated hardwired IP cores in heterogeneous MPSoCs designed with ESPAM. EURASIP J Embed Syst 2008(Article ID 726096)
37. Nikolov H, Stefanov T, Deprettere E (2008) Systematic and automated multiprocessor system design, programming, and implementation. IEEE Trans Comput-Aided Des Integr Circuits Syst 27(3):542–555
38. Nikolov H, Thompson M, Stefanov T, Pimentel A, Polstra S, Bose R, Zissulescu C, Deprettere E (2008) Daedalus: toward composable multimedia MP-SoC design. In: Proceedings of the design automation conference (DAC'08), pp 574–579
39. Pimentel A, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112
40. Pimentel A, Stefanov T, Nikolov H, Thompson M, Polstra S, Deprettere E (2008) Tool integration and interoperability challenges of a system-level design flow: a case study. In: Embedded computer systems: architectures, modeling, and simulation. Lecture notes in computer science, vol 5114. Springer, Berlin/Heidelberg, pp 167–176
41. Pimentel AD, Erbas C (2003) An IDF-based trace transformation method for communication refinement. In: Proceedings of the DAC 2003, pp 402–407
42. Pimentel AD, Polstra S, Terpstra F, van Halderen AW, Coffland JE, Hertzberger LO (2002) Towards efficient design space exploration of heterogeneous embedded media systems. In: Embedded processor design challenges, LNCS, vol 2268. Springer, pp 57–73
43. Pimentel AD, Thompson M, Polstra S, Erbas C (2006) On the calibration of abstract performance models for system-level design space exploration. In: Proceedings of the SAMOS'06, pp 71–77
44. Piscitelli R, Pimentel AD (2011) A high-level power model for MPSoC on FPGA. In: Proceedings of the IPDPS – IEEE RAW workshop 2011, pp 128–135
45. Piscitelli R, Pimentel AD (2012) A signature-based power model for MPSoC on FPGA. VLSI Design 2012:1–13
46. Rao A, Nandy SK, Nikolov H, Deprettere EF (2011) USHA: unified software and hardware architecture for video decoding. In: Proceedings of the SASP 2011, pp 30–37
47. Spasic J, Liu D, Cannella E, Stefanov T (2015) Improved hard real-time scheduling of CSDF-modeled streaming applications. In: Proceedings of the CODES+ISSS 2015, pp 65–74
48. Stefanov T, Deprettere E (2003) Deriving process networks from weakly dynamic applications in system-level design. In: Proceedings of the CODES+ISSS 2003, pp 90–96
49. Stefanov T, Kienhuis B, Deprettere E (2002) Algorithmic transformation techniques for efficient exploration of alternative application instances. In: Proceedings of the CODES 2002, pp 7–12
50. van Stralen P, Pimentel AD (2012) A SAFE approach towards early design space exploration of fault-tolerant multimedia MPSoCs. In: Proceedings of the CODES+ISSS 2012, pp 393–402
51. Thompson M, Nikolov H, Stefanov T, Pimentel A, Erbas C, Polstra S, Deprettere E (2007) A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: Proceedings of the CODES+ISSS 2007, pp 9–14
52. Turjan A, Kienhuis B, Deprettere E (2004) Translating affine nested-loop programs to process networks. In: Proceedings of the CASES 2004, pp 220–229
53. University of Amsterdam: The SESAME tool. https://csa.science.uva.nl/download/
54. Verdoolaege S (2010) Polyhedral process networks. In: Handbook of signal processing systems. Springer US, pp 931–965
55. Verdoolaege S, Bruynooghe M, Janssens G, Catthoor F (2003) Multi-dimensional incremental loop fusion for data locality. In: Proceedings of the ASAP 2003, pp 17–27
56. Verdoolaege S, Nikolov H, Stefanov T (2007) pn: a tool for improved derivation of process networks. EURASIP J Embed Syst 2007(1):1–13

57. Verdoolaege S, Seghir R, Beyls K, Loechner V, Bruynooghe M (2004) Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In: Proceedings of the CASES 2004, pp 248–258
58. Xilinx, Inc. Vivado high-level synthesis from Vivado design suite. http://www.xilinx.com/products/design-tools/vivado.html
59. Zhai J, Nikolov H, Stefanov T (2013) Mapping of streaming applications considering alternative application specifications. ACM Trans Embed Comput Syst 12(1s):34:1–34:21

SCE: System-on-Chip Environment

31

Gunar Schirner, Andreas Gerstlauer, and Rainer Dömer

Abstract

The constantly growing complexity of embedded systems is a challenge that drives the development of novel design automation techniques. System-level design can address these complexity challenges by raising the level of abstraction to jointly consider hardware and software as well as by integrating the design processes for heterogeneous system components. In this chapter, we present a comprehensive system-level design framework, the System-on-Chip Environment (SCE), which is based on the influential SpecC language and methodology. SCE implements a top-down digital system design flow based on a specify-explore-refine paradigm with support for heterogeneous target platforms consisting of custom hardware components, embedded software processors, and complex communication bus architectures. Starting from an abstract specification of the desired system, models at various levels of abstraction are automatically generated through successive stepwise refinement, ultimately resulting in a final pin- and cycle-accurate system implementation. The seamless integration of automatic model generation, estimation, and validation tools enables rapid Design Space Exploration (DSE) and efficient implementation of

G. Schirner (✉)
Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA
e-mail: schirner@ece.neu.edu

A. Gerstlauer
Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA
e-mail: gerstl@ece.utexas.edu

R. Dömer
Center for Embedded and Cyber-Physical Systems, Department of Electrical Engineering and Computer Science, The Henry Samueli School of Engineering, University of California, Irvine, CA, USA
e-mail: doemer@uci.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_31

Multi-Processor Systems-on-Chips (MPSoCs). This article provides an overview and highlights key aspects of the SCE framework from modeling and refinement to hardware and software synthesis. Using a cellphone-based example, our experimental results demonstrate the effectiveness of the SCE framework in terms of system-level exploration, hardware, and software synthesis.

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
BFM	Bus-Functional Model
BLM	Block-Level Model
CE	Communication Element
DB	Database
DCT	Discrete Cosine Transform
DSE	Design Space Exploration
ESL	Electronic System Level
FCFS	First-Come First-Serve
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDS	Hardware-Dependent Software
HLS	High-Level Synthesis
HW	Hardware
IDE	Integrated Development Environment
IP	Intellectual Property
ISS	Instruction-Set Simulator
MAC	Media Access Control
MIPS	Million Instructions Per Second
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
OOO PDES	Out-of-Order Parallel Discrete Event Simulation
OS	Operating System
PDES	Parallel Discrete Event Simulation
PE	Processing Element
PIC	Programmable Interrupt Controller
PSM	Program State Machine
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SCE	System-on-Chip Environment
SLDL	System-Level Description Language
SW	Software
TLM	Transaction-Level Model

Contents

31.1 Introduction

Designing modern Multi-Processor Systems-on-Chips (MPSoCs) becomes increasingly difficult. Challenges arise from both an increasing heterogeneity of the execution platform to meet stringent performance and power requirements, as well as from the growing application complexity demanding to integrate more, increasingly interrelated functions. Today's MPSoCs are highly heterogeneous compositions of general purpose processors, digital signal processors, graphics processors, and custom accelerators, all connected through flexible and heterogeneous interconnect systems. Designing and programming such platforms are a tremendous challenge due to heterogeneity in programming paradigms, differences in exposed parallelism, and tool suite compositions. The productivity gap between design capability and the potential in chip complexity/capacity is increasing [15]. Traditional approaches of manual implementation are tedious and error-prone as well as too time consuming to meet the shortened time-to-market demands.

One key aspect to increase productivity is to raise the level of abstraction for system design to an algorithmic level, irrespective of a later Hardware (HW)/Software (SW) split, hiding the complexity of low-level implementation details. Moving to the system level of abstraction reduces the complexity during development, enabling designers to focus on important algorithmic properties and design decisions without

being overwhelmed by the burden of low-level implementation issues. However, raising the level of design abstraction requires tool suites that are vertically integrated across all levels so as to enable a seamless codesign of software and hardware.

In this chapter, we present the System-on-Chip Environment (SCE), a vertically integrated digital system design framework based on the SpecC language and methodology [18]. SCE realizes a top-down refinement-based system design flow with support of heterogeneous target platforms consisting of custom hardware components, embedded software processors, dedicated Intellectual Property (IP) blocks, and complex communication bus architectures.

Starting off with a high-level, abstract, formal, and sound parallel programming model in SpecC language to capture the desired application behavior, the designer can define various architecture and mapping alternatives. SCE then generates Transaction-Level Models (TLMs) that realize the architecture and mapping decisions. The generated TLMs allow for detailed, simulation-based validation and performance analysis. After identifying suitable architecture and mapping candidates for an application, SCE's back-end synthesis aids in generating a cycle- and pin-accurate implementation as an interconnected set of synthesized hardware and software components down to final RTL descriptions and binary code images.

This chapter first introduces relevant related work in Sect. 31.2. It then provides a general overview of the SCE design flow in Sect. 31.3, followed by highlighting key features of the flow in more detail. Section 31.4 covers model validation with performance estimation and simulation. Section 31.5 then describes the successive model refinement for system-level architecture and communication aspects. Next, Sect. 31.6 describes the SCE software synthesis highlighting target optimization potentials, and Sect. 31.7 covers the hardware synthesis capabilities. Finally, Sect. 31.8 demonstrates the benefits of SCE with experimental results and Sect. 31.9 concludes this chapter.

31.2 Related Work

Supporting the design process has been the aim of significant research efforts with a wide range of approaches. To name a few, they range from high-level analysis and synthesis approaches that are based on specialized models of computation. Examples include POLIS [1] (Codesign Finite State Machine), DESCARTES [32] (ADF and an extended SDF), and Cortadella et al. [9] (petri nets). Integrated Development Environments (IDEs), at another end of the spectrum, typically provide limited automation support but aim to simplify manual development (e.g., Eclipse IDE [16] with its wide range of plug-in modules). Several comprehensive Electronic System Level (ESL) synthesis methodologies and tools [20] have been developed for the system-level design of heterogeneous MPSoCs. Examples include Deadalus [30], Koski [27], Metropolis [2], PeaCE/HoPES [25], SystemCoDesigner [28], and OSSS [24].

Abstract models are an important means for early prototyping and performance estimation. System-Level Description Languages (SLDLs), such as SystemC [23]

and SpecC [18], are often used for modeling of systems. At lower levels, virtual platforms allow for a detailed analysis of the system before availability of real hardware, often revealing details not available on the target [26]. While these approaches focus on modeling, simulation, and validation, they do not offer an integrated solution to generate the final implementation.

31.3 Design Flow Overview

Figure 31.1 outlines the design flow realized by the System-on-Chip Environment (SCE) [13]. The SCE flow focuses on two major steps: refinement-based system-level design space exploration in the front-end and software/hardware back-end synthesis. In the front-end exploration phase, the input specification is refined into

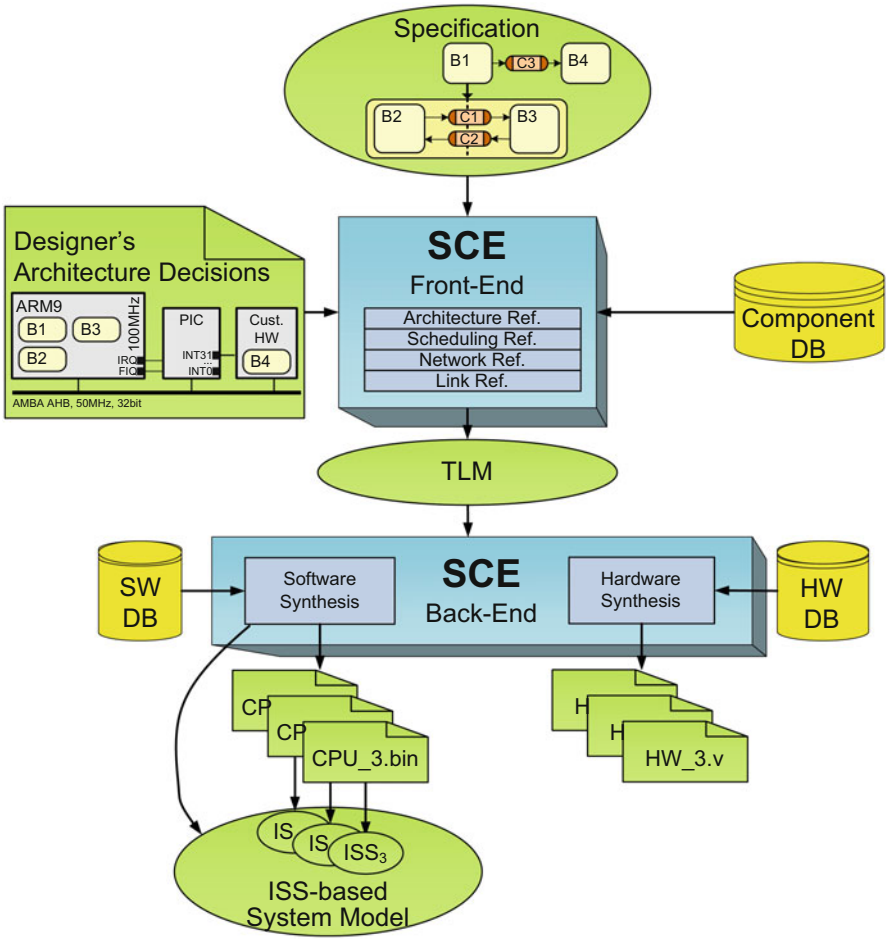


Fig. 31.1 Overall system design flow in the System-on-Chip Environment (SCE)

a TLM realizing the designer's architecture and mapping decisions for detailed performance estimation and early validation. The TLM then serves as an input for back-end synthesis, including both *software synthesis*, which automatically generates the final target software implementations for each system processor, and *hardware synthesis*, which generates the Register Transfer Level (RTL) for each custom hardware component. Both software and hardware synthesis generate matching communication stacks to realize the application distributed across the heterogeneous processing platform.

The input to the system design flow is a *specification model* containing the application captured in SpecC [18]. The specification is an abstract, parallel, platform-agnostic description of application algorithms. SpecC allows capturing a wide range of application models with an arbitrary serial-parallel composition of behaviors that communicate through abstract communication primitives for synchronous or asynchronous message passing, shared variable access, or event transfers (see Sect. 31.3.1 for an overview).

A second type of input contains the *designer's architecture decisions* including platform definition and specification mapping. For this, the designer defines the number and type of processors in the system and the topology of their communication and connectivity. In addition, the designer defines the mapping of application computation and communication onto the target platform. This includes decisions on the target software architecture (e.g., how to realize multitasking) and communication refinement decisions, such as the routing of channels over busses and the definition of essential communication parameters for each channel. For example, the user can select the synchronization scheme, such as polling or interrupt-based synchronization.

Based on the *specification model*, and the designers' *architecture decisions*, SCE then automatically generates a Transaction-Level Model (TLM) that realizes these decisions. The generation process is aided by a rich *component database* containing Processing Elements (PEs) (e.g., processors, DSPs), Communication Elements (CEs) (e.g., bridges, routers), memory components, and interconnects. In the generation process, the selected component models are instantiated and connected to create the envisioned platform. On top of this, the application (as defined in the specification) is distributed to the PEs and CEs according to the mapping decisions. Communication between PEs is refined from the standardized abstract channels in the specification down to distributed set of channels realizing the specified communication semantics on the selected platform. In order to deal with the tremendous complexity in the front-end exploration, the TLM generation is subdivided into four successive refinement steps. Each step focuses on a particular architecture aspect and by realizing, i.e., *refining*, that aspect, uncovers the next set of architecture decisions to be made (see Sect. 31.3.3).

Overall, the generated TLM captures a model of the application mapped to the envision platform realizing the architecture decisions. The application together with generated communication stacks executes behaviorally with timing back annotation on top of the timed abstract models of PEs, CEs, memories, and interconnects.

Section 31.5 illustrates the modeling in more detail. The TLM supports rapid and accurate system simulations, providing the basis for exploration, performance analysis, and debugging.

Once the designer has identified a suitable platform, the TLM also serves as input for the back-end synthesis of both software and hardware. *Software synthesis* produces a final binary image for each processor in the platform. The binary includes the application code, all drivers for communication in a heterogeneous system, as well as an off-the-shelf Real-Time Operating System (RTOS), if selected. The produced binaries can directly execute on the target processor(s) of the final hardware. For early binary validation, before availability of the real hardware, Instruction-Set Simulator (ISS)-based system models (i.e., virtual platforms) can be used. Section 31.6 discusses more details about SW synthesis. *Hardware synthesis* produces RTL for the mapped portion of the application code and the necessary communication stacks. This RTL can then be synthesized further down to a gate-level netlist and final physical realization using standard logic and back-end synthesis flows. More details about HW synthesis are described in Sect. 31.6.

The SCE flow vertically integrates from specification down to implementation through a set of consistent models, all captured in the SpecC language. This vertical integration lends itself for development of plugins at varying abstraction levels. One such example is Algo2Spec [41, 42] which automatically synthesizes a Simulink algorithm model into a SpecC specification. With this, Algo2Spec creates an extended codesign flow offering additional opportunities for algorithm designers to explore the platform suitability of algorithms and to tune algorithms to better match platform requirements (e.g., in terms of parallelism).

31.3.1 SpecC Language and PSM Model of Computation

The SCE framework is based on and closely integrated with the SpecC language and methodology [18]. It should be emphasized that this is a unique setting in which the tools, the methodology, *and* the language have been specifically created and designed together to address the needs of designing digital systems consisting of both hardware and software. In fact, the SpecC language [12] has been specifically designed to support the essential requirements for describing embedded system models at different abstraction levels. In particular, SpecC features explicit constructs and keywords for behavioral and structural hierarchy, concurrency and pipelining, synchronization and communication, exception handling, timing, and explicit state transitions. Moreover, SpecC precisely covers these requirements in an orthogonal and thus minimal manner [19].

The benefit of SpecC's language approach (in contrast to the library approach of SystemC) is the ability to parse SpecC models and unambiguously recognize the captured system modeling features. Notably, SpecC covers multiple abstraction levels, from the abstract specification model in which only functionality and design constraints are represented down to the cycle- and pin-accurate implementation

models at RTL abstraction. In contrast to commercial multi-language tools in both the hardware and software domains, the *single language* approach is a benefit since only one compiler and run-time engine must be built and maintained. In other words, the SCE framework can rely on a single-core data structure to represent the model from the beginning to the end of the design flow.

Such a homogeneous methodology does not suffer from language interfacing problems or cumbersome translations between languages with different semantics. Instead, all models are consistent, and one set of tools can be used for all models at all stages. Also, refinement tasks are merely transformations from one model into a more detailed one specified with the same language. Using a single language throughout the design process is beneficial for reuse of IP as well. Design models from the component library can be reused in the system without modification ("plug-and-play") and a new design can be inserted immediately as a library component.

The SpecC language used in the SCE design flow is based on the Program State Machine (PSM) Model of Computation (MoC). Computation and communication are separately captured using distinct language constructs. This separation enables an automatic refinement for mapping of computation to separate processing elements and establishing the communication between PEs. Computation is captured in the form of so-called behaviors, and communication is expressed in channels.

Figure 31.1 contains a graphical representation of a simple *specification model*. Boxes with rounded corners (*B1*-*B4*) symbolize behaviors. Each leaf behavior basically contains ANSI-C code, which is omitted for brevity. Behaviors can also be composed hierarchically to allow for complex structures. They can be behaviorally composed to execute in sequence, parallel, pipelined, or as state machine [19].

Behaviors are statically connected and communicate through direct point-to-point channels (*C1*, *C2*, *C3*). These channels are selected from a feature-rich set of standardized channel types, which allow for a wide range of communication mechanisms similar to what is found in an operating system. Communication primitives include synchronous and asynchronous message passing, blocking and non-blocking communication (e.g., FIFO), as well as synchronization only (e.g., semaphore, mutex, barrier).

31.3.2 Target Platform Description

The designer's target architecture decisions, as shown on the left of Fig. 31.1, describe the digital target platform. These architecture decisions include the allocation of PEs such as processors and HW components and the mapping of behaviors to PEs. The example in Fig. 31.1 shows the allocation of an ARM9 processor and one custom hardware component. The behaviors *B1*, *B2*, and *B3* are mapped to the processor. These behaviors are later wrapped into tasks, and the designer can select important task parameters, such as scheduling policies and priorities.

More generally, SCE supports distributed Multi-Processor System-on-Chip (MPSoC) target architectures as conceptually illustrated in Fig. 31.2. Target

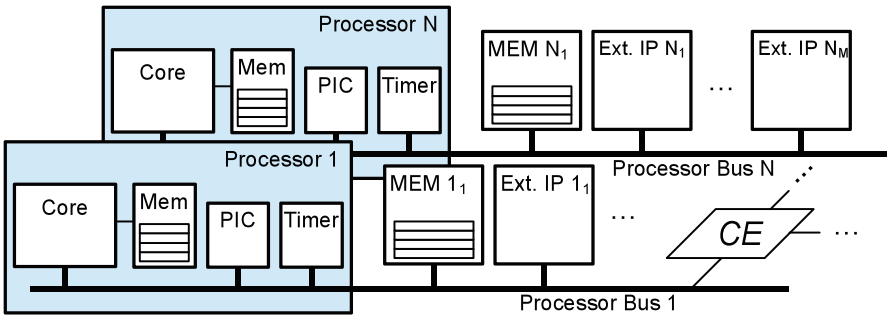


Fig. 31.2 Generic MPSoC platform targeted in SCE

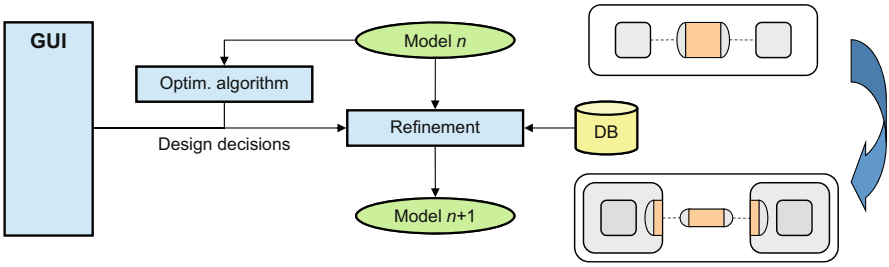


Fig. 31.3 General stepwise refinement approach in SCE

platforms can consist of a set of processors, where each processor is connected to a processor specific main bus. We assume that each processor has an associated memory, which stores the execution binaries and local variables. Additionally, we associate a customizable Programmable Interrupt Controller (PIC) and a timer with each processor. Each processor communicates with external memory (holding globally shared variables) and with hardware blocks over the processor main bus. A processor also can communicate with other processors and external IPs or memories connected to other busses through one or more Communication Elements (CEs), such as a bridges or a routers. In extension to what is shown in Fig. 31.2, we assume that the busses may be arranged as a hierarchy of busses.

31.3.3 Stepwise Refinement

The SCE framework implements a top-down design flow. SCE vertically integrates from an abstract behavioral specification down to a detailed implementation through a series of successive refinement steps. The general principle of this stepwise refinement is outlined in Fig. 31.3.

SCE's stepwise refinement separates decision-making and model refinement. Design *decisions* are primarily made by the user, entered through a GUI, or automatically determined by an optimization algorithm. Conversely, the realization of the design decision, i.e., the *refinement*, is automated. For this, a dedicated

refinement tool at a given abstraction level reads the input model ($Model_n$) and refines it following the given architecture decisions, producing the refined model ($Model_{n+1}$) which then exhibits additional implementation details that realize and represent these decisions. Each refinement tool utilizes a Database (DB) of components (such as a processor model, operating system model, etc.) to implement the decisions (such as behavior mapping or task mapping). With this separation, the tedious and error-prone part of model refinement is automated.

The SCE-internal refinement tools exchange design models in the form of a Syntax Independent Representation (SIR) [10, 39], a binary representation of a SpecC model. The SpecC compiler converts between the SpecC source code and its binary representation (the SIR). In addition, the SpecC compiler suite provides a rich Application Programming Interface (API) for convenient model transformation (such as traversing the model hierarchy, adding behaviors, or manipulating their port connectivity). Standardizing model manipulation dramatically simplifies building a refinement tool.

Four refinement levels are distinguished within SCE's front-end (Fig. 31.1). Each refinement (i.e., model transformation from $Model_n$ to $Model_{n+1}$) realizes one set of orthogonal architecture/mapping decisions and by that uncovers the set of design decisions that have to be made to guide the next refinement step (which would then create $Model_{n+2}$).

Starting from the *Specification Model*, *Architecture Refinement* realizes the mapping of behaviors to PEs and adds the necessary synchronization logic to maintain the specified execution order. Mapping parallel executing behaviors to the same PE necessitates dynamic scheduling. This prompts the user to specify scheduling parameters and priorities. *Scheduling refinement* then implements these decisions, integrating the selected RTOS model into the design, converting behaviors into tasks and setting their scheduling parameters.

The next steps then focus on communication refinement. The user is prompted to define the overall interconnect topology (bus interfaces, interconnects, communication elements). *Network refinement* realizes these decisions by inserting appropriate models from the database and ensuring proper mapping. Given the interconnect network, the final decision is mapping of channels to the interconnect structure, defining addresses and synchronization principles. *Link refinement* realizes these decisions generating the final TLM or a more detailed pin- and cycle-accurate Bus-Functional Model (BFM). Section 31.5 covers the individual refinement steps and resulting models in more detail.

31.4 Model Validation

As mentioned above, the design models used throughout the SCE flow are all represented as executable models in the SpecC language [12], regardless of the abstraction level they are specified at or refined to (The only exception is the final design model at RTL abstraction which, in addition to SpecC, can also be exported in VHDL or Verilog HDLs for hardware synthesis and the program code generated

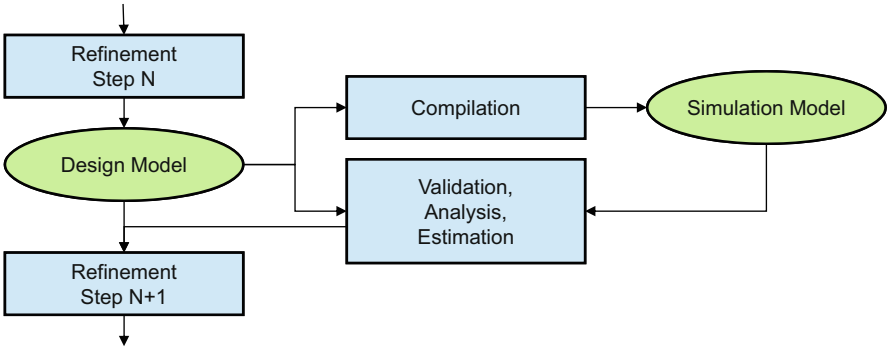


Fig. 31.4 SCE validation flow at any abstraction level: model compilation and simulation for validation, analysis, and performance metrics estimation

in ANSI-C for software synthesis.). Therefore, all models are readily executable for functional validation and evaluation through simulation. In addition, the formal nature of the models and the availability of a compiler with a comprehensive internal representation and corresponding API [10] also enables the application of formal methods for model analysis, verification, and estimation.

At any stage, as shown in Fig. 31.4, the design model together with its testbench can be fed into the compiler to generate an executable simulation model for dynamic validation (see Sect. 31.4.1 below). Alternatively, *static analysis* can be applied to the model for estimation and formal verification purposes. In contrast to dynamic simulation, where the model is executed and actual input data and specific dynamic behavior is observed, static analysis relies solely on the information stored in the SLDL source code. Here, a compiler front end reads the model code, analyzes it, and builds an Abstract Syntax Tree (AST) with control flow and typed symbol information. Based on that, static information can be extracted that is generally valid (and not just valid for a specific input). For example, where dynamic simulation shows that a model works fine for a given test set, static model analysis may prove the existence of potential deadlocks (or their absence) for *any* data input and thus show a stronger property.

In general and in practice, both static and dynamic methods are typically used together for the analysis and estimation of model metrics, in order to guide the following design decisions so that the application's requirements and goals are achieved as needed.

Note that the tasks performed in the SCE validation flow are virtually identical at any abstraction level and therefore can be performed by the same set of tools [11]. At the same time, the system designer can rely on the same methods for model validation and performance estimation, which significantly eases the learning curve for system design in SCE. Last but not least, any model can be compared against its predecessor or the golden specification model such that functional correctness and meeting of critical design goals are maintained.

31.4.1 Simulation

In system-level design, simulation is the most common form of design validation. In contrast to static analysis, simulation is dynamic and requires the design model to be executable.

In SCE, simulation is performed in two steps. First, the design model is compiled into a corresponding simulation model. More specifically, the SpecC compiler takes the design model, together with a corresponding testbench, and generates an executable program that is linked against the simulation library. The simulation library implements the execution semantics of the simulation. In particular, it maintains an event queue, advances the simulation time, and also takes care of concurrent execution and required synchronization. The generated simulation model can be run on a host computer, simulating the execution of the corresponding model. Typically, the testbench included in the simulation model supplies test vectors, checks the computed output values, and reports any problems to the user. If problems occur, a debugger can be used to set break points, interrupt the simulation, and inspect intermediate values, so that the system designer can locate and fix the errors in the model.

It should be noted that there is generally a trade-off between the run time and the accuracy of the simulation. For example, compared to the initial specification model, the refined communication model will need longer time for execution, because it may perform the communication accurately in a clock-cycle manner.

Recently, the SCE compiler and simulator have been extended to support the parallel execution of design models on multi- and many-core hosts [5, 7]. Parallel simulation, known as Parallel Discrete Event Simulation (PDES) [17] in the literature, exploits the parallelism exposed in the design model for parallel execution and thus can gain up to an order of magnitude increased simulation speed. This topic is discussed in detail in the ▸ Chap. 17, "Parallel Simulation" in this book. The advanced PDES technique specifically designed for and implemented in SCE is called Out-of-Order Parallel Discrete Event Simulation (OOO PDES) [6, 8].

31.4.2 Profiling

SCE also includes profiling tools to obtain feedback about design quality metrics. Based on a combination of static and dynamic analysis, a retargetable profiler measures a variety of metrics across various levels of abstraction [4].

Initial dynamic profiling derives design characteristics through simulation of the input model. The system designer chooses a set of target PEs, CEs, and busses from the database, and the tool then combines the obtained profiles with the characteristics of the selected components. Thus, SCE profiling is retargetable for static estimation of complete system designs in linear time without the need for time consuming re-simulation or re-profiling.

The profiling results can also be back-annotated into the model through refinement. By simulating the refined model, accurate feedback about implementation

effects can then be obtained before entering the next design stage. Since the system is only simulated once during the exploration process, the approach is fast yet accurate enough to make high-level decisions, since both static and dynamic effects are captured. Furthermore, the profiler supports multi-level, multi-metric estimation by providing relevant design quality metrics for each stage of the design process [3]. Therefore, profiling guides the user in the design process and enables rapid and early Design Space Exploration (DSE).

31.4.3 Estimation

The task of estimation is to calculate quality metrics from a design model. Although these metrics should be accurate, the main emphasis of estimation is to deliver these values quickly.

In SCE, estimated quality metrics are, for example, used for the task of architecture exploration. For instance, the trade-off between a software or a hardware solution for each behavior in the design model requires metrics for performance and cost. More specifically, the execution time and the area of each behavior are estimated for a potential hardware implementation. Also, the execution time, code size, and data size will be determined for a potential implementation in software, for each allocated processor. In addition, metrics, such as bit width and throughput, need to be determined for all channel and bus models, since these are needed for the task of communication synthesis. All these estimation results are annotated in the design model at the particular behaviors and channels. Thus, they are fed back into the synthesis flow so that this data is immediately available when needed by the synthesis algorithms.

Estimation is typically performed in form of static analysis of the design model. However, by use of profiling, estimation data can also be obtained dynamically during simulation. In SCE, profiling is used, for example, to count the execution frequency of each behavior. Based on these counter values, branching probabilities are determined, for example, for the conditional transitions in FSM behaviors. These branching probabilities are then used to estimate the average execution time for such behaviors.

Recently, the SCE profiling and estimation tools have been extended to support energy dissipation and power consumption for processing elements [33, 34]. which is essential for battery-powered mobile embedded systems. Dedicated power monitors can be inserted into the design model to observe and monitor power dissipation during the simulation. The system designer can then take these power characteristics into account when making design decisions.

31.5 Modeling and Refinement

Modeling and refinement are at the core of the SCE framework. The SCE exploration front end follows a successive, stepwise, and layer-based refinement process that employs a series of consecutive implementation and optimization passes as

Table 31.1 Summary of SCE system-level refinement steps

Ref.		Design decisions	Modeling layers
Comp.	Arch.	• Number and type of PEs and memories • Behaviors/variable to PE/memory mapping	PEs and memories (*App.*) Basic channels and memory i/fs
	Sched.	• Static behavior execution order • Dynamic scheduling policy and parameters	Operating system (*OS*) Basic channels and memory i/fs
Comm.	Net.	• Number and type of CEs and busses • Connectivity and channel routing	PE drivers (*HAL*) CE/PE transfers (*Pres./Net.*)
	Link	• Bus addressing and bus transfer modes • Bus arbitration and synchronization	PE hardware (*HW*) Bus transactions (*Link/MAC*)

described in Sect. 31.3.3. Each refinement step (see summary in Table 31.1) generates a SpecC-based Transaction-Level Model (TLM) with a level of abstraction appropriate to the refinement step. Each TLM captures an increasing amount of layered implementation detail and thus covers a different point in the simulation speed versus accuracy trade-off. These TLMs allow validation of the generated implementations while simultaneously serving as input to the back-end synthesis (see Sects. 31.7 and 31.6).

Following the general separation of computation and communication [18, 20], *computation design* is performed before *communication design* in SCE. Computation design is split into two smaller steps named *architecture refinement* and *scheduling refinement*. Communication design, similarly, is split into *network refinement* and *link refinement* (see Sect. 31.3.3). The output of the computation design, an intermediate scheduled architecture model, allows validation of the main computation mapping while exposing all required inter-PE interactions as input for further communication refinement. The final TLMs as an output of the communication design combines all computation and communication aspects of a system design in the form of PEs (modeled as SpecC behaviors) connected via busses and CEs (modeled as TLM channels and behaviors, respectively). Table 31.1 summarizes the different refinement steps including decisions made and modeling layers inserted in each step. In the following sections, we describe the computation and communication modeling and refinement steps in more detail.

31.5.1 Computation Modeling and Refinement

On the computation side, refinement transforms application behaviors in the specification into tasks and blocks partitioned and scheduled to execute in corresponding PE implementations, which are generated through a series of layer-based refinement steps. Functionality is organized into layers according to inherent conceptual dependencies. Generated output models are equally organized into layers of increasing detail. Each individual refinement step thereby introduces an additional layer of modeling and implementation detail.

SCE generally follows a host-compiled layering and modeling flow as described in ▸ Chap. 19, "Host-Compiled Simulation". For software PEs, complex, SpecC-based Operating System (OS) and processor models are automatically generated [38]. Within SCE, these models have been extended to support simulation and code generation for state-of-the-art single- and multi-core OSs and processors [31] (also discussed in ▸ Chap. 19, "Host-Compiled Simulation"). Other hardware PEs are generated as simplified variants of complete processor models that do not include all layers.

Figure 31.5 depicts the final multi-core processor model generated by SCE for the example of a dual-core ARM PE. The innermost application layer generated during architecture refinement encapsulates the specification behaviors mapped onto the processor. During scheduling refinement, these behaviors are converted into user tasks running on top of scheduling services provided by an OS layer and OS model (realized as a SpecC OS channel). As described in ▸ Chap. 19, "Host-Compiled Simulation", the OS channel provides typical services for OS and task management, synchronization, inter-process communication and timing via a canonical OS API that is later translated into real OS calls during back-end software synthesis (see Sect. 31.6). In the process of converting specification behaviors into OS tasks, the code is also back-annotated with estimated delays as described in more detail in ▸ Chap. 19, "Host-Compiled Simulation".

In addition to basic OS and task services, the OS layer also provides high-level communication functionality for sending and receiving inter-processor application-level messages via an underlying Hardware Abstraction Layer (HAL). During architecture and scheduling refinement, only basic models for channel adapters, drivers, interrupt tasks, and interrupt handlers are inserted as templates into the OS layer and HAL. These templates are later filled with actual code during link refinement, where HALs with pre-written, canonical models for Media Access Control (MAC) and bus/TLM interfaces are taken out of SCE's component database. Together, the OS and HAL ultimately provide and realize timing-accurate models of communication protocol stacks that transform application-level message channels all the way down to corresponding transaction-level bus accesses plus interrupt-driven or polling-based synchronizations, if required. See Sect. 31.5.2 for more details.

Finally, the HAL is encapsulated into a Hardware (HW) layer that models external bus communication via a bus channel. The HW layer of the processor model also emulates monitoring of processor interrupt signals and associated processor exceptions to model a general, timing-accurate multi-core interrupt handling logic and chain. From the hardware side, core-specific interrupt requests are generated by a generic multi-core interrupt controller (GIC) model, which manages the distribution and dispatch of interrupt signals to processor cores. The HW layer in turn contains core-specific interrupt detection logic (shown as interrupt interfaces in Fig. 31.5) that triggers interrupt execution in the HAL. To emulate processor suspension, hardware-triggered interrupts are modeled as special, high-priority interrupt handlers associated with each interrupt source. Thus, when an interrupt is detected by a core's interrupt interface, the HAL will notify the OS model (via a

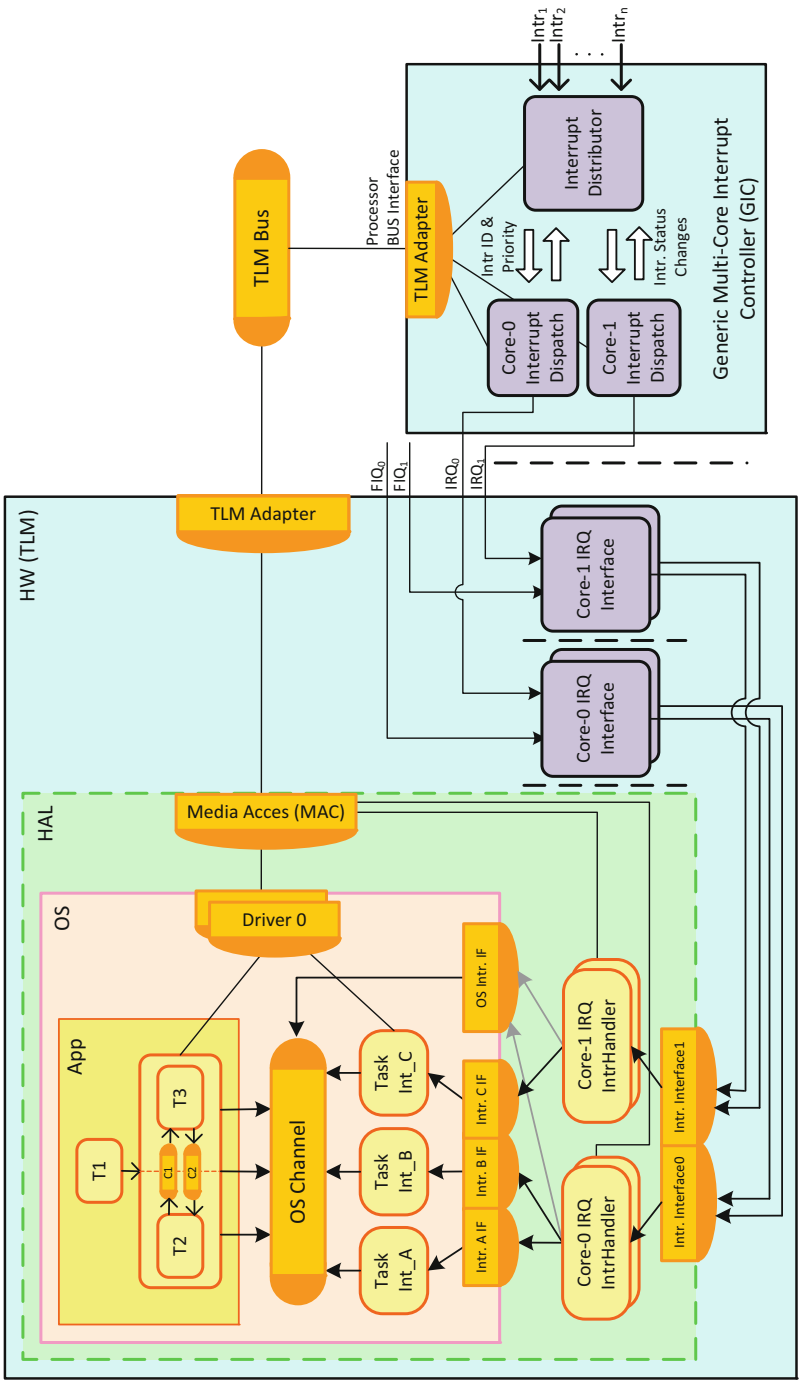


Fig. 31.5 General layer-based multi-core processor model

special OS interrupt interface) to preempt and switch execution to the corresponding interrupt handler in the HAL. The interrupt handler, as generated during link refinement, can then in turn communicate with the GIC and, through associated OS layer interfaces, trigger corresponding secondary interrupt-specific OS tasks (as shown for the example of interrupts *A*, *B*, and *C* in Fig. 31.5).

Overall, the application, OS, HAL, and HW layers all combined constitute the host-compiled SpecC processor model. This processor model is then integrated into a standard TLM backplane for simulation in the context of an overall multiprocessor system environment.

31.5.2 Communication Modeling and Refinement

In the model generated after architecture and scheduling refinement, PEs still communicate via high-level primitives at the message-passing level, where a PE's HAL and HW layers are still left out. Network and link refinement then transform such abstract application-level communication channels all the way down to transactions over busses or other (shared) communication media. In the process, HAL and HW layers are added to PEs, and optimized code is generated for drivers, interrupt handlers, and bus transactors inserted into software and hardware PEs, respectively. All driver, interrupt handling, and transactor code is back-annotated with estimated delay information to provide an overall timing-accurate simulation of communication overheads.

Similar to the computation side, communication modeling and refinement follow a layer-based organization adapted and derived from the ISO/OSI 7-layer model [21]. At the output, low-level TLMs that include protocol stacks and inter-PE communication at varying levels of detail depending on the number of included layers are automatically generated. These TLMs are synthesized into actual software or hardware during back-end synthesis.

Figure 31.6 shows the general organization of a final TLM including all layers as generated by SCE for the example of a system architecture with two PEs, *PE0* and *PE1*, representing a software processor and a hardware accelerator, respectively. The two PEs are connected via two busses and an intermediate transducer *T*. Application behaviors *P1* and *P2* within each PE communicate with each other using abstract *send()* or *receive()* primitives. Additional protocol layers are inserted into PEs to realize all such channel communication over external busses. On the software side, this protocol functionality is inserted into corresponding processor model layers as described in Sect. 31.5.1 above.

Network refinement transforms end-to-end messages exchanged over abstract communication channels into point-to-point packet transfers over individual bus segments. In the process, additional CEs such as protocol-level bridges or network-level transducers are inserted as necessary to interconnect, translate, and forward packets between different bus segments or communication media. Communication layers inserted during network refinement include a presentation layer for channel-specific data conversion (*c1*, *c2*, and *c3*) and a network and transport layer (*Net*) for packeting, routing, and end-to-end synchronization.

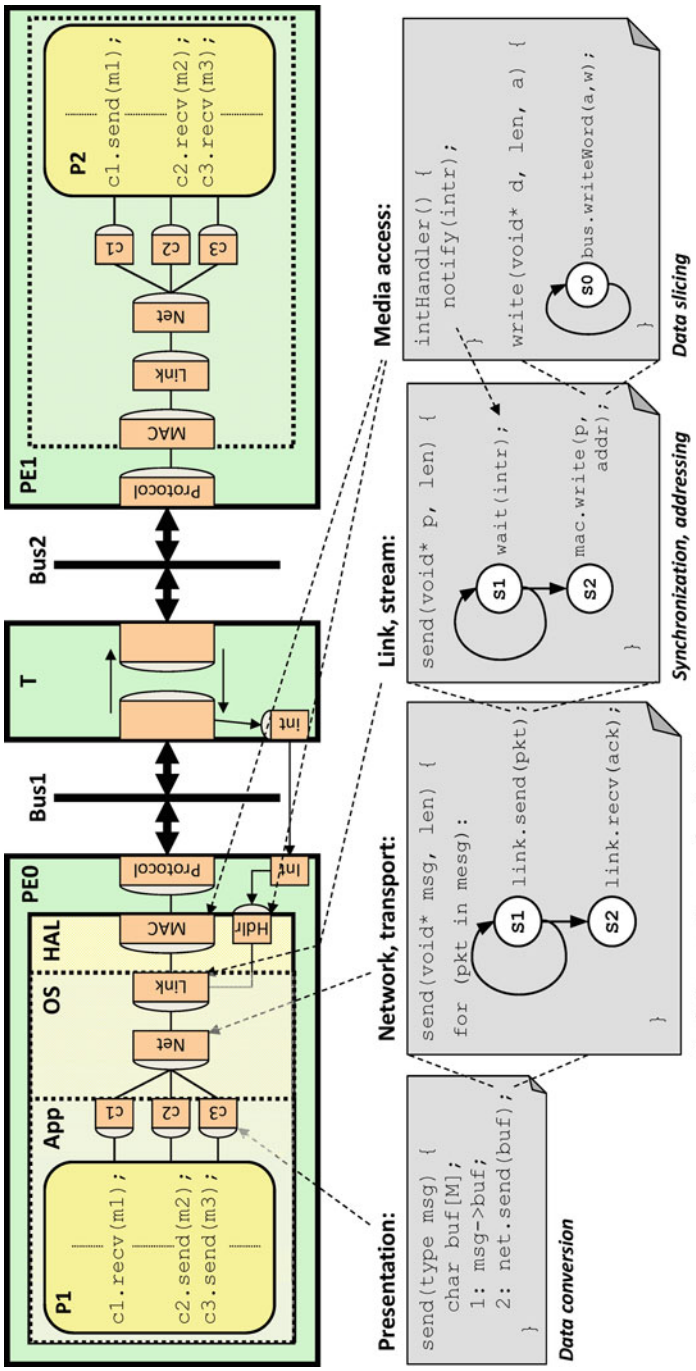


Fig. 31.6 General layer-based transaction-level communication model

Subsequent link refinement then transforms all point-to-point transfers with each bus segment into actual bus transactions. It inserts a link layer (*Link*) for addressing and interrupt- or polling-based synchronization, a MAC layer for data slicing, and a final *Protocol* layer implementing actual bus state machines. In case of interrupt-driven synchronization, this further includes hardware-level interrupt detection and generation logic (*Int*) as well as software-level interrupt handlers (*Hdlr*) as described in previous sections.

TLMs of the system can be generated to model inter-PE communication at any of these levels. For example, bus channels connecting PEs in a protocol TLM describe communication at the level of individual bus transactions. By contrast, faster but less accurate MAC or link TLMs describes communication at the level of larger unsynchronized or synchronized whole-packet transfers [35]. Finally, a pin- and bus cycle-accurate BFM can be generated by also including a low-level protocol layer describing individual events of bus transactions on each wire.

Basic network and link refinement generates unoptimized code for protocol stacks following a strict layer-based organization as shown in Fig. 31.6. Recent extensions to SCE allow protocol code to be further optimized specifically for efficient back-end synthesis [29]. Protocol stack optimizations flatten and merge basic communication layers above the MAC and apply back-end-specific cross-layer optimizations for message merging, protocol fusion, and interrupt hoisting. Depending on the target PE, fused upper layers are later synthesized into optimized software drivers or hardware transactors. By contrast, MAC and protocol layers are usually provided as pre-designed software or hardware IP in the back-end synthesis databases. In case of hardware PEs, higher-level protocol functionality can be further coupled with and synthesized into low-level bus state machines using protocol IP generators that support corresponding customizations.

31.6 Software Synthesis

Once the designer has settled on a set of architecture decisions and is satisfied with the performance of the generated TLM, the back-end synthesis can be invoked as illustrated on the bottom portion of the flow overview in Fig. 31.1. This includes both software synthesis and hardware synthesis.

Software synthesis [37] uses the generated TLM (i.e., the output of link refinement; see Sect. 31.5) as input and produces embedded code. It is invoked for each programmable PE generating a SW stack matching the overall system. For this, the synthesis approach implements the application modeled in the TLM, which is captured in the SpecC SLDL, on a target processor. The TLM includes SLDL-specific keywords and concepts, such as behaviors, events, channels, and port mappings. To realize these SLDL concepts in target software, one approach would be to replicate the SLDL simulation environment directly on the target platform. This, however, potentially results in overhead for performance and code size. Instead, our software synthesis directly generates embedded ANSI-C code out of the SLDL to achieve compact and efficient code.

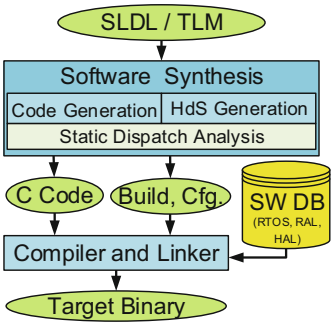


Fig. 31.7 Software generation flow in SCE

We divide *software synthesis* into *code generation* and *Hardware-Dependent Software (HDS) generation*, shown in Fig. 31.7. Code generation deals with the code inside each task and generates flat ANSI-C code out of the hierarchical model captured in the SLDL. Meanwhile, HDS generation creates code for processor-internal and processor-external communication, adjusts for multitasking, and eventually generates configuration/build files (e.g., Makefiles) for the cross compilation process. Software synthesis is supported by a SW database, which contains static target-specific artifacts, such as an operating system, that will be linked in when creating the final binary.

31.6.1 Software Code Generation

Code generation [40] produces sequential ANSI-C code for each task within a programmable PE. It translates the hierarchical composition of behaviors in SpecC into flat C-code consisting of functions and data structures. For SLDL features not natively present in the targeted ANSI-C language (e.g., port/interface concept, hierarchical composition), code generation realizes these SLDL features out of available ANSI-C constructs.

While ANSI-C was chosen as it is widely used in the embedded context and has a rich compiler support, some challenges emerge as ANSI-C does not have language constructs to realize object-oriented programming. As one example, SpecC behaviors offer local variables visible within an instance of the behavior. This could be realized with a class. ANSI-C, however, does not provide an encapsulation/scope for class-local storage as it lacks the class concept. To overcome this limitation, all behaviors' local variables are added to a behavior-representing structure, and all accesses to behavior-local variables are replaced with accesses to the corresponding member of the behavior-representing structure.

Similar challenges appear for methods that are exposed as an interface on the behavior's port. Here, each port also becomes a member of the behavior-representing structure. In addition, the structure includes a virtual function table (VTABLE) pointing to the implementing methods. All calls to these methods are replaced with function calls through the VTABLE entries. In principle, embedded

code generation realizes some basic object-oriented concepts on top of ANSI-C. As such, it solves similar issues as C++ to C compilers that translate a class hierarchy into flat ANSI-C code.

31.6.2 Hardware-Dependent Software Generation

The second portion, HDS generation [36, 37], generates code for processor-internal and processor-external communication, including drivers and synchronization (polling or interrupt). It also generates code to execute multiple tasks on the same processor, realizing the task mapping defined in the computation refinement (Sect. 31.3.3). To create the complete binary SW image, it finally generates configuration and build files (e.g., Makefile) which select and configure database components.

To realize the HDS generation, we distinguish code that is only platform specific and code that is both platform and application specific. The former, i.e., platform-specific code, is instantiated from the SW database. This DB includes a selection of RTOSs to provide multitasking and a basic HAL for canonical access to common platform resources. During HDS generation, code for instantiating the selected RTOS is generated, and the RTOS is configured toward the application requirements. In order to unify the access to a wide variety of RTOSs, each RTOS in the DB is accompanied by an RTOS Abstraction Layer (RAL). The RAL abstracts from the particular RTOS's function names and parameters. To ensure a generic API, we investigated different RTOS APIs (uCOS-II, vx-Works, eCos, ITRON, POSIX) and chose common primitives for task scheduling, communication, and synchronization. Along similar lines, the HAL provides canonical access to common platform resources, which we assume to be present in all target platforms (such as timers and an interrupt controller). The canonical APIs (as realized by RAL and HAL) limit the interdependency between HDS generation and the selected target architecture, thus making HDS generation more scalable.

Code that is both application and platform specific is produced by the HDS generation. This includes code for multitasking, internal communication, and external communication. For multitasking, application-specific code is generated to instantiate and control the tasks as defined the TLM (e.g., *T1, T2, and T3* in Fig. 31.5). Internal communication, which occurs within the same PE, is realized along the same lines. Channels *C1* and *C2* in Fig. 31.5 are examples of PE-internal communication. These channels are replaced with an implementation on top of the RAL of the selected RTOS (e.g., using semaphores and memcpy). External communication and synchronization occur between PEs. As part of communication refinement (Sect. 31.5.2), external communication has been refined into a set of stacked channels as visualized in Fig. 31.6. To realize the specified communication, HDS generation traverses the TLM from the innermost layer (e.g., *PE0*'s presentation layer with the stacked channels *c1*, *c2*, and *c3* in Fig. 31.6) and generates a matching driver stack. The generated driver stack utilizes the RAL (e.g., for interrupt registration, handling, and synchronization) as well as the HAL (e.g., for accessing the processor bus).

Combining the outputs of code generation and HDS generation yields all code for a PE. Using a cross-compiler, the final target binary is created. The SW synthesis process repeats for each software PE in the TLM. The produced binaries can directly execute on the target processor(s) of the final hardware. For early binary validation, before availability of the real hardware, *virtual platforms* can be used. To facilitate this step, our software synthesis also includes a model refinement step that converts the input TLM into an ISS-based system model. For this, SW synthesis removes the processor model for which it has generated the SW earlier (i.e., everything inside the *HAL* layer of *PE0* in Fig. 31.6) and replaces it with an ISS from the database. The integrated ISS instance can then run the generated binary including all platform-specific interactions (e.g., for communication with custom hardware) for early binary validation and further codevelopment.

31.6.3 Software Optimization

Software synthesis offers several optimization opportunities across code generation and HDS generation. One example is static dispatch analysis, which, if permissible, eliminates the overhead of virtual function calls. Virtual function calls are common in object-oriented code to allow multiple implementations for the same interface. This is very frequently used in layered implementation of complex systems (such as in our layer-based realization of communication between PEs). The overhead of virtual functions appears in many languages, such as C++, SpecC, SystemC, as well as in our generated code. Generally, the overhead for a virtual function call itself is low (2 cycles [14]). More importantly, however, this indirection hinders inlining optimizations. Especially when the callee function has only minimal computation, the virtual function call overhead becomes quite significant on embedded platforms.

To improve the performance and quality of SLDL synthesized embedded SW, we have enhanced our embedded ANSI-C code synthesis with a dispatch-type analysis to reduce/eliminate this overhead [43]. Our approach utilizes the fact that the complete model (with all connectivity) is known during SW synthesis and no SW is linked later on top of the generated code. Hence, static dispatch-type analysis can determine if a direct call is possible (i.e., the call target can be determined statically) and replace a virtual call with a direct function call.

31.7 Hardware Synthesis

Hardware synthesis refines models of custom hardware processors in the TLM down to complete RTL implementations following an extended High-Level Synthesis (HLS) design step [29]. The generated RTL can then be further synthesized down to a gate-level netlist and final physical realization using standard logic and back-end synthesis flows. SCE integrates a hardware synthesis flow that combines existing commercial, off-the-shelf HLS tools for synthesis of computation with the capability

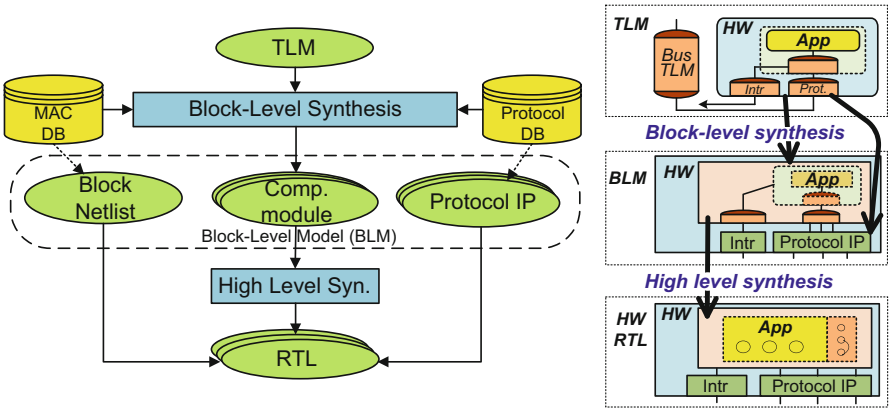


Fig. 31.8 Hardware synthesis flow in SCE

to synthesize communication interfaces and hardware bus transactors supporting a wide range of optimized target implementations [29].

Figure 31.8 shows the overall flow of hardware synthesis in SCE. We follow a three-step methodology to transform the TLM into a SystemC/C++-based Block-Level Model (BLM), further synthesize computation blocks in the BLM down to cycle-accurate RTL using a traditional HLS tool, and then perform logic synthesis to generate a final gate-level netlist.

31.7.1 Block-Level Synthesis

Block-level synthesis refines the TLM generated during front-end computation and communication refinement (see Sect. 31.5) down to block-level modules, protocol IPs, and a block-level netlist integrating all of these together. Low-level blocks for protocol layer IPs are thereby taken directly out of a protocol database in pre-written RTL form. By contrast, other higher-layer blocks are converted from SpecC into synthesizable SystemC or C++ code to be further synthesized and as supported by existing HLS tools.

Most HLS tools can synthesize multiple single-threaded C++/SystemC modules with the capability to stitch blocks together. However, they cannot automatically partition preexisting code. In order to provide a general approach that can be easily adapted to different HLS back ends, we partition the code into separately synthesized modules that are integrated through our own netlisting engine. In the process, each computation or communication behavior in the TLM is converted into a separate, synthesizable hardware block. After block partitioning, TLM communication stacks are inlined into each accessing computation block. This enables computation/communication cooptimizations in the following HLS step, with the scheduling freedom to overlap computation with communication and to perform general, joint optimizations, such as resource sharing or parallelization.

Communication layers in the TLM connect to each other and to computation behaviors via an interface mechanism. Such high-level functional interface and variable ports are converted into low-level wire, register or First-In First-Out (FIFO) interfaces between blocks in the BLM as supported by the underlying HLS engine. At the lowest MAC and protocol levels, thin Media Access Control (MAC) layer implementations inserted from a hardware database provide the glue logic between application-specific higher layers and pre-written bus protocol IPs. Synthesizable MAC database implementations thereby replace canonical MAC layer models and interfaces in the TLM with code that provides equivalent, canonical MAC-level bus interfaces to higher layers while internally translating each transaction into corresponding pin- and wire-level interactions necessary to interface with a target-specific bus protocol IP component.

31.7.2 Protocol IP Generation

Each bus protocol layer in the TLM is replaced with an actual protocol IP from the protocol database. Based on parameters, such as the type and number of ports, a protocol generator thereby creates a custom IP block from pre-written RTL templates in the protocol database to implement external bus protocols and, depending on the synchronization mechanism selected and defined in the TLM, either interrupt generation or polling logic. The internal interface of bus protocol IPs is designed to match associated protocol wrappers in the MAC database, which, when synthesized together with other blocks, will realize appropriate pin- and wire-level interactions with the generated IP.

31.7.3 RTL Netlisting and Synthesis

The connectivity of all blocks, IPs, and external ports is converted into an block-based RTL netlist to complete the block-level synthesis step. As indicated in Sect. 31.7.1 above, in the process high-level functional variable and channel interfaces between modules are converted into wires, registers, and glue logic connecting the pin-level FIFO ports of synthesized blocks and protocol IPs. After block-level synthesis, computation blocks in the generated BLM are further synthesized down to cycle-accurate RTL descriptions using an external HLS tool. Generated RTL blocks, RTL bus protocol IPs, and the block netlist are then synthesized into a gate-level netlist using a traditional logic synthesis tool.

Utilizing tools and databases along with a HLS engine, our flow can be easily adapted to different HLS back ends and synthesis targets. Our current SCE setup includes support for Mentor (formerly Calypto) Catapult, and Xilinx Vivado HLS tools targeting ASICs or FPGAs with Mentor Precision or Xilinx ISE as logic synthesis back ends.

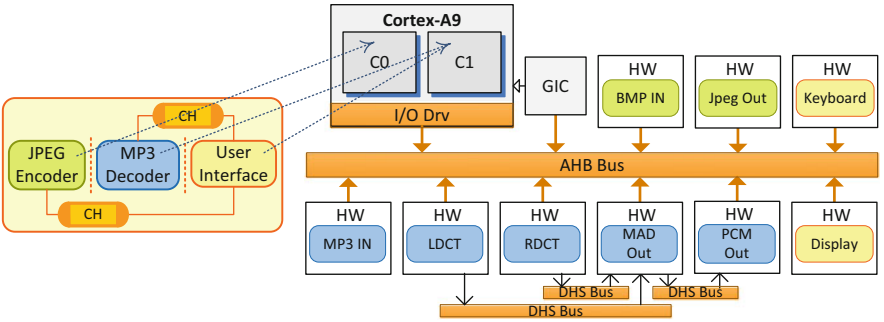


Fig. 31.9 Cellphone baseband example

31.8 Experimental Results

We demonstrate the benefits of SCE for fast and accurate system-level Design Space Exploration (DSE) and synthesis as applied to a cellphone baseband example running concurrent Motion-JPEG (M-JPEG), MP3, and user interface tasks on a dual-core 650 MHz Cortex-A9 platform. The overall architecture of the system is shown in Fig. 31.9 [31]. The MP3 decoder and JPEG encoder use optional hardware accelerators to perform audio decoding or Discrete Cosine Transform (DCT) and quantization acceleration, respectively. Tasks communicate with external hardware and the rest of the system via an AHB bus and 12 interrupts. In this experiment, MP3 decodes 13 frames at a bitrate of 384 kbit/s, and JPEG encodes 10 frames of a movie with standard 352×288 resolution at a rate of 30 frames/s.

We explored a wide range of architectures by applying different OS and processor configurations, including mapping of M-JPEG tasks and interrupts to two different cores (C0 and C1) in a dual-core architecture. We explored both First-Come First-Serve (FCFS)/FIFO and priority (Prty) scheduling. In dual-core architectures with task-attached interrupts, application tasks are distributed among two cores and a task, and its associated interrupts are mapped to the same core. By contrast, dual-core architectures with a core-attached interrupt always handle all interrupts on C1. Figure 31.10 summarizes the average frame delays and average absolute errors in frame delays as well as maximum error bars of MP3 and JPEG tasks. Frame delays and frame delay errors are reported both for high- and low-level TLM simulations of the platform at a link and MAC level of communication abstraction, respectively, as compared to an ISS simulation. Task delays were back-annotated at the function level from measurements taken on the ISS. Moreover, average Linux context-switch overhead was measured and back-annotated into the OS model.

As can be seen, the best MP3 performance is achieved when a higher priority is assigned to MP3, or MP3 and JPEG are running on separate cores. In other configurations, average MP3 frame delay is close to its deadline boundary (i.e., 26.1 ms).

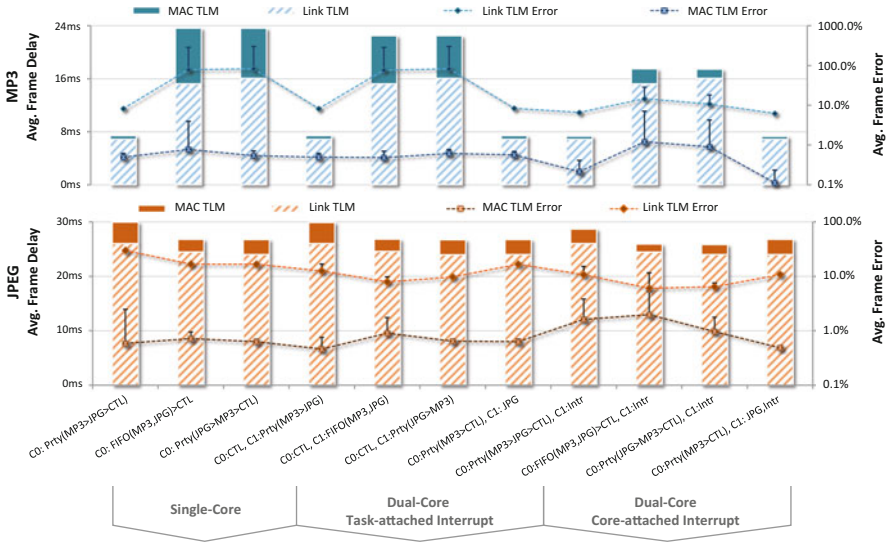


Fig. 31.10 Cellphone design space exploration

Minimized JPEG delay is obtained from configurations with FIFO scheduling, when JPEG has higher priority or when it runs on a separate core. All combined, explorations confirm that shortest-job-first or rate-monotonic scheduling guarantee that MP3 and JPEG meet their performance requirements. Overall, optimized MP3 and JPEG performance is achieved when tasks run on separate cores. Finally, by mapping all interrupts to a separate core (C1), we only see slight performance benefits in MP3 and JPEG delays.

Overall, the SCE system design front end provides an efficient platform for rapid, early, and accurate DSE. On average, low-level MAC TLMs generated by SCE simulate with 1,400 Million Instructions Per Second (MIPS) at less than 1% error. A model at higher level of abstraction that does not account for synchronization overhead can achieve even higher simulation speeds, but errors can reach as high as 100% in some configurations where inaccuracies lead to MP3 and JPEG tasks executing in the wrong order.

With the definition of architecture and mapping in place, the SCE system design back end is activated to generate both software and hardware for each processing element.

31.8.1 Software Synthesis

SW synthesis has been used to create the implementation for the cellphone example described above. To provide more insight into the optimization potential offered by SW synthesis, we will look into the static dispatch analysis, which can convert

virtual function calls to direct calls and by that avoid the VTABLE indirection overhead. For this, we look more closely into the JPEG compression task of the cellphone platform.

We anticipate the most optimization potential in the generated drivers as they use a layered approach and contain very little own computation. To gain more insight into the influence of communication, we vary the communication granularity. The input (*BMP In* to *JPEG*) can either operate on individual pixels or coarser on a whole pixel row. The output (*JPEG* to *JPEG Out*) is configurable to operate on individual bytes or more efficiently by using a queue for buffered communication (queue size 256 bytes). If selected, the queue is mapped to the processor. This increases efficiency as the processor writes with small operations (mostly bytes) and the JPEG output retrieves larger blocks. The coarser granularity reduces the number of transactions by using fewer, yet larger transactions. This results in fewer calls into the layered drivers. Thus, we anticipate less optimization potential through static dispatch analysis with fewer, larger transactions. Please note that varying the communication granularity is only done for analysis purposes. A real product would use the coarsest granularity to reduce overhead.

Table 31.2 quantifies the static dispatch analysis optimization for the JPEG encoder example in its four configurations. In addition to the execution time, it shows the speedup as well as the number of virtual function calls converted into direct calls.

Table 31.2 confirms the expectations. With the coarsest communication granularity (row, queue), the JPEG encoder executes the fastest (39.6 ms), while the finest granularity (pixel, byte) executes the slowest (57 ms). The number of MAC driver calls gives an indication for the slowdown. With 41 K calls, the finest granularity needs almost 4 times as many calls for the same amount of data as the coarsest granularity. All configurations significantly benefit from static dispatch analysis. The speedup through converting to direct calls ranges from 12.4% for the coarsest to 16.1% for the finest grained communication. Although fewer port method calls are converted to direct calls in the fine-grained case (81 instead of 89), the drivers are called much more frequently (41 K calls instead of 11 K calls) leading to the larger speed up through our optimization. Irrespective of the communication granularity, our approach offers tremendous benefits as it eliminates the virtual function calls for the generated embedded SW while also improving code readability.

Table 31.2 Static dispatch analysis optimization

Input	Output	Exec. time [ms]	Exec. time (opt) [ms]	Improvement	# Direct calls	# MAC drv. calls
Row	Queue	39.64	34.74	12.4%	89	11469
	Byte	42.69	36.89	13.6%	85	16560
Pixel	Queue	54.01	46.31	14.3%	85	36351
	Byte	57.06	47.86	16.1%	81	41442

31.8.2 Hardware Synthesis

To demonstrate the hardware back end of SCE, we have applied the flow to synthesis of hardware accelerators for the JPEG encoder subsystem of the cellphone example [29]. Specifically, the DCT and Quantize blocks (DCT-Q) in the JPEG encoder are mapped into hardware while the rest of the application functionality executes on the application processor. We used Mentor Catapult as HLS back end, targeting an ARM+FPGA platform consisting of a Freescale i.MX21 applications processor and a Xilinx Spartan-3 Field-Programmable Gate Array (FPGA) communicating over Freescale's EIM bus.

Using our fully automated hardware synthesis flow, we were able to synthesize the TLM into RTL ready for further FPGA download within minutes, yielding substantial productivity gains compared to a manual design process. On the processor side, software was automatically synthesized, combined with driver code, and cross-compiled into a Linux executable (see Sect. 31.8.1 above).

We synthesized computation blocks and communication channels using polling- or interrupt-based synchronization targeting a 50MHz clock frequency. We applied protocol stack and protocol coupling optimizations as described in Sect. 31.5.2 during synthesis. We compare FPGA resource usage and hardware latency of the optimized design (POPT) against an unoptimized design (NOPT) and a purely manual implementation of communication interfaces (MAN). For a fair comparison, the manual design utilizes the same computation block and the same firmware code synthesized by a HLS tool and SCE, respectively. We evaluated end-to-end hardware latency by instrumenting the JPEG encoder software to record the average turnaround times over 180 DCT-Q invocations, including all communication and synchronization overhead. Area and latency results of synthesized accelerator PEs are summarized in Table 31.3.

We can observe that the latency of final, optimized hardware synthesized with our flow is always significantly less than in a manual design. In contrast to a manual design in which communication and computation blocks are designed separately to manage complexities, our approach is able to perform cooptimizations across computation and communication boundaries [29]. In the DCT-Q case, data is processed strictly in the order it is received and sent, which allows computation and communication to be pipelined and scheduled in parallel. This can overlap and hide communication latencies behind computation delays. Due to its complexity

Table 31.3 JPEG hardware synthesis results

Synch.	Opt.	LUTs	FFs	Logic score	Mem [bytes]	Latency
Intr.	NOPT	5334	3197	8531	1024	59.9 ms
	POPT	4091	2046	6137 (−28%)	1024 (0%)	59.7 ms (0%)
	MAN	3284	2182	5466 (−36%)	1024 (0%)	89.0 ms (49%)
Poll.	NOPT	5407	3302	8709	1024	9.4 ms
	POPT	4133	2341	6474 (−26%)	1024 (0%)	9.8 ms (4%)
	MAN	3281	2179	5460 (−37%)	1024 (0%)	28.4 ms (203%)

and non-modularity, such optimizations are typically not applied in manual designs. However, a naive, unoptimized realization of such computation/communication codesign can lead to a large increase in total area. The proposed protocol stack optimizations prove to be efficient in reducing this area overhead through resource sharing, which results in up to average 26% logic reduction compared to an unoptimized design. Overall quality of results is comparable to or better than a manual design, where on average 2.2 times improvement in latency can be achieved.

31.9 Conclusions

Designing modern embedded systems is increasingly challenging due to hardware complexity (e.g., increasing heterogeneity), functional complexity (increasingly complex, integrated features), more stringent nonfunctional requirements, all while reducing the time to market. Addressing the complexity challenges requires raising the level of abstraction to jointly consider digital hardware and software while paving a path to automated implementation.

In this chapter, we have presented our comprehensive system-level design framework, the System-on-Chip Environment (SCE). SCE realizes a top-down codesign flow, from an abstract, functional specification captured in the SpecC language down to synthesized hardware and software. It supports a wide range of heterogeneous target platforms consisting of custom hardware components, embedded software processors, and complex communication bus architectures.

This chapter has provided an overview and highlighted key aspects of the SCE framework: the underlying language and simulation principles as well as the layer-based modeling and refinement. It also provided an overview of the software and hardware synthesis processes. We have illustrated the capabilities of our SCE codesign framework through a cellphone example targeted to a heterogeneous target architecture.

The focus of SCE is on the modeling, refinement, and implementation synthesis process. In its current form, all design decisions have to be entered manually through SCE's Graphical User Interface (GUI) or via an external scripting interface. This complements other system-level design tools that have a focus on automated Design Space Exploration (DSE) as presented in the book Part 3, "Design Space Exploration". In order to leverage different strengths of existing tools [20], we have combined SCE with other academic DSE engines to provide a comprehensive, seamless, and fully automated system-level synthesis solution all the way from high-level, data-flow-based system specifications down to concrete HW and SW implementations for generic MPSoC platforms [22]. This provides a proof-of-concept realization of a complete system-level synthesis solution. The authors thank Dongwook Lee and Parisa Razaghi for their help in preparing this manuscript and the anonymous reviewers for their valuable suggestions on its improvement. The authors express gratitude to all members of the SpecC team who have contributed to SCE over the years, namely Samar Abdi, David Berner, Lukai Cai, Pramod Chandraiah, Che-Wei Chang, Vincent Chang, Weiwei Chen, Alexan- der Gluhak,

Yitao Guo, Xu Han, Ran Hao, Eric Johnson, Jon Kleinsmith, Deepak Mishra, Guantao Liu, Junyu Peng, Gautam Sachdeva, Yasaman Samei, Dongwan Shin, Sanyuan Tang, Ines Viskic, Qiang Xie, Haobo Yu, Pei Zhang, Jiaxing Zhang, Shuqing Zhao, Jianwen Zhu. Last but not least, the authors thank Daniel D. Gajski for his visionary leadership.

References

1. Balarin F, Chiodo M, Giusto P, Hsieh H, Jurecska A, Lavagno L, Passerone C, Sangiovanni-Vincentelli A, Sentovich E, Suzuki K, Tabbara B (1997) Hardware-software co-design of embedded systems: the POLIS approach. Kluwer Academic Publishers, Boston
2. Balarin F, Watanabe Y, Hsieh H, Lavagno L, Passerone C, Sangiovanni-Vincentelli A (2003) Metropolis: an integrated electronic system design environment. Trans Comput 36(4):45–52
3. Cai L, Gerstlauer A, Gajski D (2005) Multi-metric and multi-entity characterization of application for early system design exploration. In: IEEE/ACM Asia and South Pacific design automation conference (ASP-DAC)
4. Cai L, Gerstlauer A, Gajski DD (2004) Retargetable profiling for rapid, early system-level design space exploration. In: Proceedings of the design automation conference (DAC), San Diego
5. Chen W, Han X, Chang CW, Dömer R (2013) Advances in parallel discrete event simulation for electronic system-level design. IEEE Des Test Comput 30(1):45–54
6. Chen W, Han X, Chang CW, Liu G, Dömer R (2014) Out-of-order parallel discrete event simulation for transaction level models. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 33(12):1859–1872. doi:10.1109/TCAD.2014.2356469
7. Chen W, Han X, Dömer R (2011) Multi-core simulation of transaction level models using the system-on-chip environment. IEEE Des Test Comput 28(3):20–31
8. Chen W, Han X, Dömer R (2012) Out-of-order parallel simulation for ESL design. In: Proceedings of the design, automation and test in Europe conference and exhibition (DATE)
9. Cortadella J, Kondratyev A, Lavagno L, Massot M, Moral S, Passerone, C, Watanabe Y, Sangiovanni-Vincentelli A (2000) Task generation and compile time scheduling for mixed data-control embedded software. In: Proceedings of the design automation conference (DAC), Los Angeles
10. Dömer R (1999) The SpecC internal representation. Technical report, information and computer science. University of California, Irvine. SpecC V 2.0.3
11. Dömer R (2000) System-level modeling and design with the SpecC language. Ph.D. thesis, University of Dortmund
12. Dömer R, Gerstlauer A, Gajski D (2002) SpecC language reference manual, version 2.0. SpecC Technology Open Consortium. http://www.specc.org
13. Dömer R, Gerstlauer A, Peng J, Shin D, Cai L, Yu H, Abdi S, Gajski D (2008) System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design. EURASIP J Embed Syst 2008:647953
14. Driesen K, Hölzle U (1996) The direct cost of virtual function calls in C++. SIGPLAN Not 31(10):306–323. doi:10.1145/236338.236369
15. Ecker W, Müller W, Dömer R (2009) Hardware dependent software: introduction and overview. In: Ecker W, Müller W, Dömer R (eds) Hardware dependent software: principles and practice. Springer, Berlin
16. Eclipse Foundation. Eclipse. http://www.eclipse.org/
17. Fujimoto R (1990) Parallel discrete event simulation. Commun ACM 33(10):30–53
18. Gajski DD, Zhu J, Dömer R, Gerstlauer A, Zhao S (2000) SpecC: specification language and design methodology. Kluwer Academic Publishers, Boston

19. Gerstlauer A, Dömer R, Peng J, Gajski DD (2001) System design: a practical guide with SpecC. Kluwer Academic Publishers, Boston
20. Gerstlauer A, Haubelt C, Pimentel A, Stefanov T, Gajski D, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput Aided Des Integr Circuits Syst 28(10):1517–1530
21. Gerstlauer A, Shin D, Peng J, Dömer R, Gajski DD (2007) Automatic layer-based generation of system-on-chip bus communication models. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 26(9):1676–1687
22. Gladigau J, Gerstlauer A, Haubelt C, Streubühr M, Teich J (2011) Automatic system-level synthesis: from formal application models to generic bus-based MPSoCs. Trans High-Perform Embed Archit Compil (Transactions on HiPEAC) 5(4):1–22
23. Grötker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic Publishers, Dordrecht
24. Grüttner K (2015) Application mapping and communication synthesis for object-oriented platform-based design. Ph.D. thesis, Carl von Ossietzky University Oldenburg
25. Ha S, Kim S, Lee C, Yi Y, Kwon S, Joo YP (2008) Peace: a hardware-software codesign environment for multimedia embedded systems. ACM Trans Des Autom Electron Syst 12(3):24:1–24:25. doi:10.1145/1255456.1255461
26. Hong S, Yoo S, Lee S, Lee S, Nam HJ, Yoo BS, Hwang J, Song D, Kim J, Kim J, Jin H, Choi KM, Kong JT, Eo S (2006) Creation and utilization of a virtual platform for embedded software optimization: an industrial case study, Seoul
27. Kangas T, Kukkala P, Orsila H, Salminen E, Hännikäinen M, Hämäläinen TD, Riihimäki J, Kuusilinna K (2006) UML-based multiprocessor SoC design framework. ACM Trans Embed Comput Syst 5(2):281–320. doi:10.1145/1151074.1151077
28. Keinert J, Streubühr M, Schlichter T, Falk J, Gladigau J, Haubelt C, Teich J, Meredith M (2009) SystemCoDesigner – an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. Trans Des Autom Electron Syst 14(1):1:1–1:23
29. Lee D, Park H, Gerstlauer A (2012) Synthesis of optimized hardware transactors from abstract communication specifications. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS)
30. Nikolov H, Thompson M, Stefanov T, Pimentel A, Polstra S, Bose R, Zissulescu C, Deprettere E (2008) Daedalus: toward composable multimedia MP-SoC design. In: Proceedings of the design automation conference (DAC 2008), pp 574–579
31. Razaghi P, Gerstlauer A (2014) Host-compiled multi-core system simulation for early real-time performance evaluation. ACM Trans Embed Comput Syst (TECS) 13(5s):166:1–166:26
32. Ritz S, Pankert M, Zivojnvic V, Meyr H (1993) High-level software synthesis for the design of communication systems. IEEE J Sel Areas Commun 11(3)348–358. doi:10.1109/49.219550
33. Samei Y, Dömer R (2014) Automated estimation of power consumption for rapid system level design. In: Proceedings of the IEEE international performance computing and communications conference
34. Samei Y, Dömer R (2014) Powermonitor: a versatile API for automated power-aware ESL design. In: Proceedings of the forum on specification and design languages (FDL)
35. Schirner G, Dömer R (2009) Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. ACM Trans Embed Comput Syst (TECS) 8(1):4:1–4:29
36. Schirner G, Dömer R, Gerstlauer A (2009) High-level development, modeling and automatic generation of hardware-dependent software. In: Ecker W, Müller W, Dömer R (eds) Hardware dependent software: principles and practice. Springer, Berlin
37. Schirner G, Gerstlauer A, Dömer R (2008) Automatic generation of hardware dependent software for MPSoCs from abstract system specifications. In: Proceedings of the design automation conference. Asia and South Pacific (ASPDAC), Seoul
38. Schirner G, Gerstlauer A, Dömer R (2010) Fast and accurate processor models for efficient MPSoC design. ACM Trans Des Autom Electron Syst (TODAES) 15(2):10:1–10:26

39. Viskic I, Dömer R (2006) A flexible, syntax independent representation (SIR) for system level design models. In: Proceedings of the EUROMICRO conference on digital system design (DSD), pp 288–294
40. Yu H, Dömer R, Gajski D (2004) Embedded software generation from system level design languages. In: Proceedings of the design automation conference. Asia and South Pacific (ASPDAC), Yokohama
41. Zhang J, Schirner G (2014) Automatic specification granularity tuning for design space exploration. In: Proceedings of the ACM/IEEE conference on design, automation & test in Europe (DATE), Dresden. doi:10.7873/DATE.2014.227
42. Zhang J, Schirner G (2015) Towards closing the specification gap by integrating algorithm-level and system-level design. Des Autom Embed Syst (DAEM) 19:389–419. Springer. doi:10.1007/s10617-015-9161-1
43. Zhang J, Tang S, Schirner G (2015) Reducing dynamic dispatch overhead (DDO) of SLDL-synthesized embedded software. In: Asia and South Pacific design automation conference (ASPDAC), Chiba

Metamodeling and Code Generation in the Hardware/Software Interface Domain

32

Wolfgang Ecker and Johannes Schreiner

Abstract

In the HW/SW interface domain, specification of memory architecture and software-accessible hardware registers are both relevant for the implementation of hardware and the firmware running on it. Automated code generation of both HW and SW artifacts from a shared data source is a well-established method to ensure consistency. Metamodeling is a key technology to ease such code generation and to formalize the data structures target code is generated from. While this can be utilized for a wide range of automation and generation tasks, it is particularly useful for bridging the HW/SW design gap.

Metamodeling is the basis for the construction of large model-driven automation solutions that go far beyond simple code generation solutions. Based on the formalization metamodels provide, models can be incrementally transformed and combined to create more refined models for particular design tasks. IP-XACT and UML/SysML can be utilized within the scope of metamodeling. The utilization of these standards and the development of custom metamodels – targeted to specific design tasks – have proven to be highly successful and promise large potential for further productivity increase.

Acronyms

AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
API	Application Programming Interface
AST	Abstract Syntax Tree
AXI	Advanced eXtensible Interface
BNF	Backus-Naur Form
CIM	Computation Independent Model

W. Ecker (✉) • J. Schreiner
Infineon Technologies AG, Neubiberg, Germany
e-mail: wolfgang.ecker@infineon.com; johannes.schreiner@infineon.com

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_32

CPU	Central Processing Unit
DMA	Direct Memory Access
EBNF	Extended Backus-Naur Form
EDA	Electronic Design Automation
EMF	Eclipse Modeling-Framework
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High-Level Synthesis
HTML	Hypertext Markup Language
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
JSON	JavaScript Object Notation
MDA	Model-Driven Architecture
MoC	Model of Computation
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Program State Machine
RTL	Register Transfer Level
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SW	Software
TLM	Transaction-Level Model
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema
XSLT	Extensible Stylesheet Language Transformations

Contents

32.1 Introduction

Productivity increases in the design of embedded systems always built on the idea of predesigning modules from smaller components, providing an abstract model and other views and packaging all that together for use in a higher-level design environment.

First, semi-custom design prepacked transistors to logic gates and provided models with logic functions, propagation delay, and a graphical representation symbolizing the functionality of the gate. Typical representatives for prepacked gates are AND gates with two, three, or four inputs and one output or a D flip-flop with clock, reset, and data (normally called `D`) input and one output (normally called `Q`). This packing also enabled the use of gates in a schematic editor which provides a graphical view of the model. This stage of development permitted increasingly complex designs which could no longer be manually handled on the lower levels of abstraction. The final layouts were therefore done by fully automatic place and route tools.

Next, gates were prepacked to RT components, associated with register-transfer functionality and with untimed or clock-related timing. The packing was enriched with schematic views, with operators or program constructs and their mapping to the RT components. A good example for this is an adder which supports various sized inputs and the associated information that a "+"-operator can be mapped to it. The essential achievement of this abstraction is that RTL synthesis tools can map RTL descriptions to gate-level netlists in an automated way.

Further pursuing this approach, IP components were introduced that were pre-implemented in RTL and associated with more abstract TLM models to enable early and efficient simulations of multimillion – if not billion – transistor chips. Although that approach is now about 15 years old, this technique is not fully established. If established, generation of TLM and RTL top levels – i.e., abstract model and implementation – from a single source model is not a widely used approach. This is one of the main reasons for delayed introduction of automated IP-based design with TLM models: automation as provided by layout and RTL synthesis tools is not available. Further abstraction – except for some prepacked subsystems – is not widely used today [21].

Solely relying on the reuse of prepackaged items makes it very hard to implement innovative products since innovation is limited to novel combination of pre-implemented items. To address this limitation, RTL synthesis provided an additional

abstraction: the ability to describe the behavior of a design using sequential constructs known from programming languages following a specific coding style. By applying a mapping – called inference in the RTL domain – these sequential constructs are mapped to RTL netlists and RTL primitive components that are then further optimized and synthesized as described above. For example, an if-statement causes the insertion of multiplexers for all signals assigned in the statement blocks.

The increase of productivity in RTL-synthesis provided for a wide range of digital designs through behavioral constructs could however not be repeated. High-Level Synthesis (HLS) tools, state machine synthesizers, or processor generation tools – to name only some – could improve productivity only in very limited fields of application. Moreover, reuse and composition of IP components do not give the productivity increase that is often claimed since they help to design chips with a lot of transistors, yet the transistors still need plenty of custom firmware and software on top of them to work properly.

If a single tool cannot provide system-level automation – i.e., automation beyond implementation level – for a wide range of applications, why not use a tool suite with tools that follow the same concept, interact, and together provide a wide range of automation [8]. This approach is exemplified by Office suites which provide a collection of tools for presentations, text documents, spreadsheets, project management, and much more.

However, simple scripting as successfully used by many designers (see, e.g., [27]) is too expensive to provide a sufficient number of tools at an acceptable cost and effort. Metamodeling techniques [7] provide a substantial measure to dramatically shorten the building time of such tools. Therefore metamodeling is one key technology to enable system-level automation via tool suites and to ease the interaction between tools which are part of these tool suites.

The goal of this book section is to introduce metamodeling in general and to show how it helps to increase productivity around the HW/SW interface. In the first subsection, we introduce the general concept of metamodeling and show early metamodeling technologies. Afterward, we give a formal definition of a metamodel illustrating the formalization and giving an idea, on how metamodels can be used in a formalized design process. Finally, we describe some metamodeling techniques in use, show the idea of automatic view generation around the HW/SW interface, and illustrate the basic structure of a metamodeling framework.

32.2 What Is Metamodeling About

First of all, metamodeling is different from other modeling approaches and also uses the term *model* differently than, e.g., in semi-custom or RTL design. This is further elaborated in Sect. 32.2.2. In metamodeling, a model describes an entity, mostly an intended design by its properties, its sub-entities, and the relationships between them.

32.2.1 A First Example

32.2.1.1 A Simplified View on the HW/SW Interface

Figure 32.1 shows a very simple metamodel of an IP's register interface. Comparable register interfaces are a key component of generic HW/SW interfaces. The HW/SW interface works basically as follows:

- By writing a value to the base address of the IP plus an internal offset, a value is passed from software to hardware. From the SW point of view, this is similar to writing to a memory cell. Therefore, SW can treat those addresses as special variables. Additionally hardware can be attached to the register, e.g., to trigger actions when the register is accessed or when a specific value is written. Connection of the registers to the IP-HW is done via wires. IP-HW then processes the values of those wires.
- Storing a value in the bitfield from the hardware side or giving access to HW wires via the bitfield provides a way for the software to read a value from hardware. After having read the register the bitfield resides in, the software can further process the value. Similar to writing of values, SW handles the values read from the IP-HW like values from a memory. They can thus be treated like special variables in the software context.

In the following we focus on this basic mechanism, ignoring that there are additional possibilities for HW/SW interfaces such as CPU accessible special function registers, interrupts, or DMA request lines. When taking a closer look at the conceptual description above and considering the abstraction levels mentioned in the introduction, it becomes clear that several levels of abstraction with the same or different Model of Computation are bridged:

- The software side follows primarily a sequential, control flow-oriented execution order. The software is mostly developed in C and C++ – although assembler code

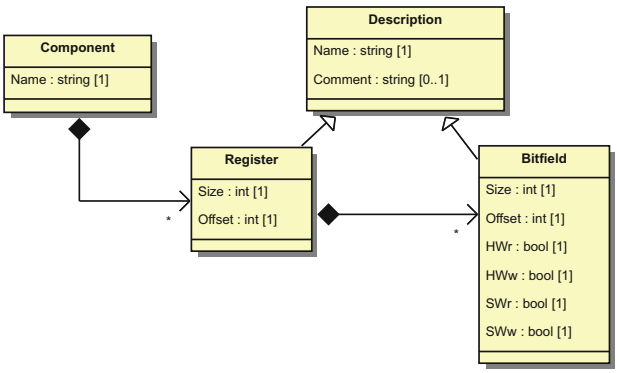


Fig. 32.1 Simple metamodel of register interfaces

is still used. These descriptions do not contain timing information and individual threads of execution have no degree of parallelism.
- The hardware side follows several concepts. Depending on the level of abstraction the hardware is observed from, it uses different modeling languages:
  - When observed from the gate-level perspective, connected timed primitives mostly described in Verilog are used.
  - On RTL, additional synchronous control flow or data flow is an appropriate view, typically modeled in VHDL or SystemVerilog.
  - TLM communicating processes are mostly coded in SystemC.

32.2.1.2 A First Metamodel

Instead of trying to make one model composed of sub-models, each following an own model of computation and being interlinked with a multi-domain formalism as proposed in [12], metamodels follow another idea: metamodels identify involved entities and define their attributes and relations. Further, metamodels also define constraints such as types, valid values, or valid multiplicity.

Figure 32.1 shows the definitions for the key entities involved in a HW/SW interface: `Component`, `Register`, and `Bitfield`. `Component` is the root node. In this model, it has a required string attribute `Name` and an unlimited number of registers. The latter is shown by the association arrow and the multiplicity `*`. Each `Register` has the mandatory attribute `Offset`, specifying the offset of the register in the address space of the component. Since a register must have this attribute, its multiplicity is set to `1`. In addition, the offset must be a number which is defined by the type `int` of the attribute. Similarly, the register has a definition of its `Size`.

Finally, a register has one or more bitfields, again shown by the association arrow pointing at `Bitfield` and the multiplicity `1..*`. A `Bitfield` has an offset `Offset` in the bit space of the register and a `Size`. Their type is `int` since both must be an integer number. Both have multiplicity `1` since they are mandatory attributes for a bitfield. To specify how a bitfield can be accessed, our register metamodel has four mandatory Boolean attributes `SWreadable`, `HWreadable`, `SWwritable`, and `HWwritable`.

Figure 32.1 also shows two Unified Modeling Language (UML) generalization arrows. These arrows point from the entities `Register` and `Bitfield` to `Description`. They indicate that the entities `Register` and `Bitfield` acquire all attributes and associations from the arrow target `Description`. Since `Description` has the mandatory string attribute `Name` and the optional string attribute `Comment`, `Register` and `Bitfield` have these attributes too. Of course, `Register` and `Bitfield` acquire all properties of these attributes as well. Thus, inheritance does not provide additional measures to describe entities; however, it simplifies and structures the description of their properties.

If you noticed that Fig. 32.1 resembles a UML class diagram, you are right: Fig. 32.1 was captured with DoUML, an open-source UML editor [24]. Although metamodels and UML class diagrams have many things in common, they are not the same. As we will see later, metamodels are used in UML to define class diagrams.

32.2.1.3 A First Model

With a metamodel at hand, a so-called *model* can be built that meets the constraints of the metamodel. In other words, legal instances of the metamodel can be built. The metamodeling technique follows the idea of separation of model and view, RTL models or SystemC-TLM models are thus called views in metamodeling terms. Before we discuss views in Sect. 32.2.1.4, we take a closer look at a model.

Figure 32.2 shows an example of a model specified in a graphical way using a UML object diagram. The model describes one instance of `Component`, its two registers `R1` and `R2` and its four bitfields `B1` to `B4`. All `Name` attributes are set, i.e., the multiplicity constraints imposed from the metamodel are satisfied here. No `Comment` is set here which is legal since its multiplicity defined in the metamodel makes the attribute optional. All other mandatory attributes are set. The attributes `Offset` and `Size` are set to integer values, and `HWr`, `HWw`, `SWr`, and `SWw` are set to the Boolean values `True` or `False`.

There are further constraints originating from the semantics underlying the domain modeled here. These constraints are also met, which are easily comprehensible when looking at the diagrams. For example, every bitfield can either be read or written from each design domain, i.e., at least readable or the writable from software, as well as at least readable or writable from hardware. Further, the size of the bitfields is smaller than the size of the register. Since these additional constraints cannot be shown in a graphical way, they can be annotated in the tool capturing the metamodel either using a specific constraint language such as Object Constraint Language (OCL) (see [33]) or a programming language.

The model describes a component called `Simple`. This component has two 16-bit registers `R0` and `R1`. `R0` is at the relative address 0 and `R1` at the relative address 1. In contrast to what one might assume, the register does not describe a storage element in our model. Instead, it only describes an addressable shell. Bitfields are responsible for holding data and therefore the access rights are specified here. In our case, if a bitfield cannot be written from the SW side, the value intended to be written via the register access is simply ignored. If it cannot be read, then the value

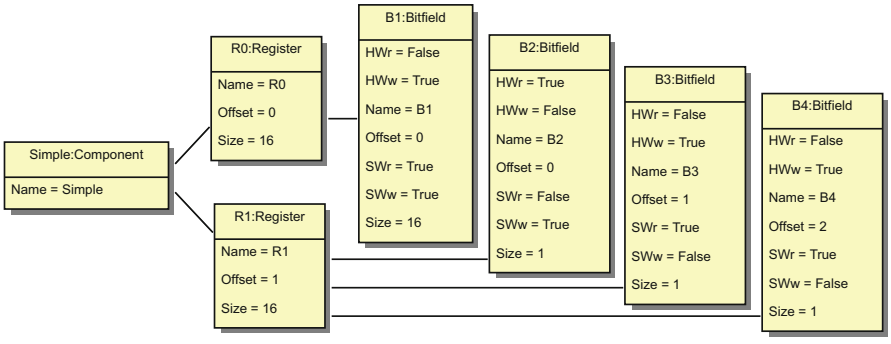


Fig. 32.2 Simple model of a register interface

0 is returned. On the HW side, read and write access flags determine if there is a line from the register field to the HW core of the IP. The value written to HW is stored in a temporary register and the read value is directly taken from the IP core. These are all assumptions underlying our simplistic model. Industrial strength models such as IP-XACT (see Sect. 32.4.2.1) offer a wider range of possibilities here.

In addition to the UML diagram specification, models can be specified in many other ways, using, e.g., XML, JSON, or spreadsheets. It is important to assert that there is only one place where – and only one way how – the model is defined. This is called *single source* approach and prevents inconsistencies. It is especially important on the HW/SW interface since several design domains are bridged here.

32.2.1.4 First Views

Documentation

For this model, several views exist in the design process. One of them is documentation. A tabular representation of such a documentation view is shown in Table 32.1. This view is used, e.g., by the verification engineers validating the interface, the software and hardware engineers making the interface, and by the customer developing software for the product the IP is integrated in. This table and all views shown here are simplified to provide a better perspective on the overall methodology. For an industrial documentation, please take a look, e.g., at [17].

RTL Code

A possible RTL view of our model is shown in Fig. 32.3. The bus interface is assumed to consist only of `Addr`, `DataIn`, `DataOut`, `En`, and `Wr`. `En` is `"1"` if the IP is accessed and `Wr` is `"1"` if a register should be written. Bus and register are assumed to have the same clock and reset signal and there is no pipelining or other delay on the bus. Of course, component metamodels such as the aforementioned IP-XACT have possibilities to define more sophisticated buses (e.g., AXI, AHB, or APB) which then lead to more complex bus interfaces. However, they follow the same basic concept that is introduced here.

The first process `SW_WRITE` is responsible for write accesses from the SW side and the second process `SW_READ` for read accesses. The first process is also responsible for the inference of the synchronous memory elements needed for

Table 32.1 Simple component register documentation table

Component : *Simple*									
Register			Bitfield						
Name	Offset	Size	Name	Offset	Size	HWr	HWw	SWr	SWw
R0	0	16	*B1*	0	16	*False*	*True*	*True*	*True*
R1	1	16	*B2*	0	1	*True*	*False*	*False*	*True*
			B3	1	1	*False*	*True*	*True*	*False*
			B4	2	1	*False*	*True*	*True*	*False*

```
1  entity SimpleRegs
2     port(
3        -- Generic Interface
4        Clk, Rst: in std_ulogic;
5        -- \acs{CPU} Interface
6        Addr    : in  std_logic_vector(15 downto 0);
7        DataIn  : in  std_logic_vector(15 downto 0);
8        En, Wr  : in  std_logic;
9        DataOut : out std_logic_vector(15 downto 0);
10       -- \acs{HW} Interface
11       R0_B1_i : in  std_logic_vector(15 downto 0);
12       R0_B1_o : out std_logic_vector(15 downto 0);
13       R1_B2_o : out std_logic;
14       R1_B3_i : in  std_logic;
15       R1_B4_i : in  std_logic
16    );
17 end entity regs;
18
19 architecture \acs{RTL} of SimpleRegs is
20 begin
21    SW_WRITE: process( Clk, Rst )
22       if Rst = '1' then
23          R1_B2_o <= '0'; R0_B1_o <= (others => '0');
24       elsif rising_edge( Clk ) then
25          if En = '1' and Wr = '1' then
26             case Addr is
27             when B"0000_0000_0000_0000" => R0_B1_o <= DataIn;
28             when B"0000_0000_0000_0001" => R1_B2_o <= DataIn( 1 );
29             end case;
30          end if;
31       endif;
32    end process;
33    SW_READ: process( En, Addr )
34    begin
35       if En = '1' and Wr = '0'then
36          case Addr is
37          when B"0000_0000_0000_0000" =>
38             DataOut <= R0_B1_i;
39          when B"0000_0000_0000_0001" =>
40             DataOut <= "0" & R1_B3_i & R1_B4_i & B"0_0000_0000_0000";
41          when others => DataOut <= X"0000";
42          end case;
43       else
44           DataOut <= X"0000";
45       end if;
46    end process;
47 end architecture RTL;
```

Fig. 32.3 VHDL file

storing the bitfield values written by the software. The second process is responsible for the multiplexers needed to provide the right value to SW via the port `DataOut`. The bitfields which cannot be accessed from SW are simply omitted. Similarly, only those bitfields marked to be SW readable contribute to the value to be passed to SW. All bits of a register with no bitfield contribution are filled with `0` as shown in line 40.

If a bitfield is written and read from the same party – as the case for bitfield `B1` of our example – then the IP-HW is responsible to feed either the same value back or a different one. It is worth noting that it is therefore not guaranteed that the same value which is written is read back.

For each bitfield that is read by the hardware, a port is created. Port `R0_B1_o` in line 12 illustrates the format of the names of these ports: They consist of the name of the register the bitfield is part of (`R0`), the bitfield's name (`B1`), and a character indicating the direction (`o`). These elements are concatenated to the port name with underscores (_).

Correspondingly, ports are created for bitfields written by the hardware. Finally, for each bitfield's port, a type is selected which both matches with its size and can be merged to a legal value of `DataOut`.

C-Header File

Figure 32.4 shows a possible firmware view of the HW/SW interface. For each register, a struct with elements representing the bitfields is created. The size of a bitfield is defined using the "`:`" operator followed by the size. The type is always `uint16_t` indicating a 16-bit wide unsigned integer. The keyword volatile indicates that the bitfields may be modified outside the software. The compiler thus cannot cache the values, e.g., in a CPU register or optimize the number of read accesses and must access the raw bitfield every time it is used by the software (i.e., generally the C-code).

The registers are then combined to an overall register interface `reg_t` using another struct. Assuming an instance `SimpleInst0` and a pointer called `SimpleInst0Ptr` (see Line 17), then bitfield `b2` can be accessed by:

```
SimpleInst0Ptr->r1.b2
```

However, different C compilers won't accept this coding style since "`:`" is not generally supported. Further, `unit16_t` may not result in the intended result.

```
1  struct r0_t {
2     volatile uint16_t b1   : 16;
3  };
4
5  struct r1_t {
6     volatile uint16_t b2   : 1;
7     volatile uint16_t b3   : 1;
8     volatile uint16_t b4   : 1;
9     const uint32_t unused : 13;
10 };
11
12 struct reg_t {
13    volatile r0_t r0;
14    volatile r1_t r1;
15 };
16
17 reg_t *SimpleInst0Ptr;
```

Fig. 32.4 C-header file

```
1  struct reg_t {
2     volatile bus_t r0;
3     volatile bus_t r1;
4  };
5
6  reg_t *SimpleInst0Ptr;
7
8  inline bool GetR1B3(reg_t *rp) {
9     return (bool) ( (rp->r1) >> 1 ) & 0x01 );
10 }
```

Fig. 32.5 Another C-header file

To address this, a style similar to Fig. 32.5 might be needed. Here, inline functions are used to access the bitfields. Line 8 shows how access to the bitfields is provided using a combination of shift, mask, and type cast.

There may be even more styles and variants of C's HW/SW interface view since `bus_t` and `bool` may not be supported. For example, macros could be used instead of inline functions. For an overview of different coding styles, see, e.g., Chap. 5 "Hardware/Software Interface" of [6]. The important aspect of the metamodeling approach is the guaranteed consistency of all these views which is ensured by code generation from the same specification source.

So far, we have only seen a model as instance of a metamodel and as an *additional* view to all the existing views. Before identifying the case for metamodeling in Sect. 32.2.4, let's discuss the metamodeling terminology and take a look at the evolution of metamodeling over the last decades.

32.2.2 Terminology

32.2.2.1 Metamodel

So far, we simply accepted the term *metamodel* as something that constrains a model but we did not dig deeper into the prefix *meta*. As opposed to metaphysics, which is not a specific domain of physics but a branch of philosophy, metamodeling is a term from computer science.

The relationship between both terms is that *meta* stands for *beyond*. While metaphysics deals with questions about *the fundamental nature of being* [13], metamodeling deals with *fundamental concept of a model*. Therefore, a *metamodel* models the domain of a model. In other words, a *metamodel* is a model of a model.

This definition is in line with the things that were already said about metamodeling. Our simple metamodel example models the domain of the HW/SW interface. It is a guide – and also a constraint – for each specific model of the HW/SW interface of a component.

32.2.2.2 Metametamodel

There is however also a metamodeling domain. The models of this domain are *metamodels*. These models of the metamodeling domain in turn have their own

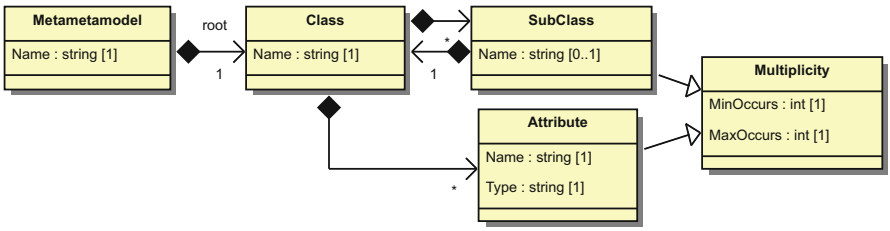


Fig. 32.6 Meta-metamodel

metamodels describing them. In other words, each model of the metamodeling domain (a *metamodel*) is an instance of its own metamodel; the metamodel thus has a metamodel. When viewed from the modeling domain, each model m has a metamodel mm. As this metamodel mm also has a metamodel, the latter is the meta-metamodel mmm of m.

An example of a meta-metamodel is shown in Fig. 32.6. It describes that a metamodel has a root class which has other classes and attributes, both of them in any multiplicity. Therefore, a class references a container for a subclass, which potentially redefines the name and specifies the multiplicity. For simplicity, base classes are not shown here. This does not reduce the expressiveness of the meta-metamodel since they do not contribute to the modeling possibilities as such.

Interestingly, this meta-metamodel can be defined using the formalism that is also used for the metamodel. This concept is not unusual in computer science. For example, the BNF grammar can be defined using an BNF grammar itself (see [34]). This is a hint that there might not be a *meta-meta-metamodel*, although there is some research to find this even more basic model.

Figure 32.7 shows the component metamodel from Fig. 32.1 as an instance of the meta-metamodel shown in Fig. 32.7. The object `ComponentMM` is the root node in the meta-metamodel instance named `ComponentMM`. Its only associated object `Register` of type `Class` is the root node of the metamodel. Its associated objects `Offset`, `Size`, and `Name` of type `Attribute` specify the attributes of `Register`. The object Register also has an associated `Subclass` object defining that the associated object `Bitfield` of type `Class`. The multiplicity `1..*` of this association is represented by `MaxOccurs=1` and `MinOccurs=-1`. The `Bitfield` class has seven attribute objects associated with, three of which are not completely shown.

32.2.2.3 Metamodeling Layers

The relationships of the introduced artifacts, views, models, metamodels, and meta-metamodels, are pictured in Fig. 32.8. Here, we see that the artifacts are labeled from $M0$ to $M3$, a terminology introduced by OMG (see [22]). We also see that the higher numbered artifacts define the structure of their directly lower numbered artifact. In turn, the lower labeled artifacts are an instance of – or in other words comply to – their directly higher numbered artifact. The M0-M1 relation differs as

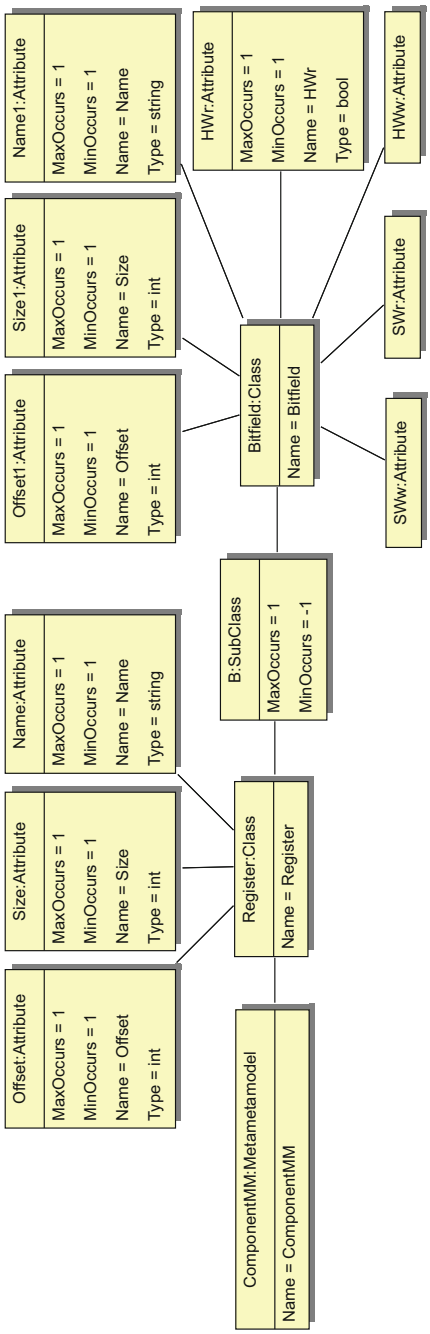


Fig. 32.7 Meta-metamodel instance: component

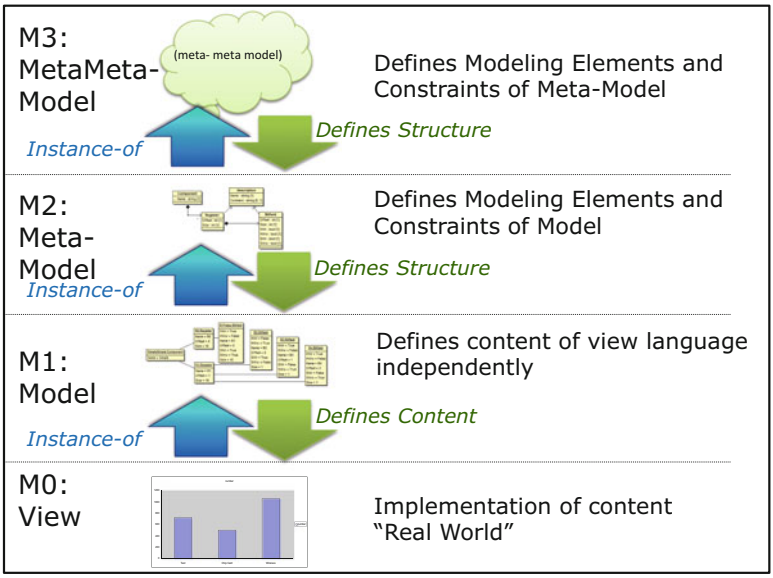


Fig. 32.8 View, model, metamodel, and meta-metamodel

it is a content wise and not a structural dependency. In some cases, this dependency may also be structural, as the introduction of model-to-model transformations in Sect. 32.4.2.3 shows.

Although the depicted designation of layers is widely accepted, there are two alternative approaches. A small set of publications follow the idea that HW design deals with models, i.e., what we call a view is in their definition a model. Consequently, what we refer to as models, metamodel, and meta-metamodels becomes metamodel, meta-metamodel, and meta-meta-metamodel. This definition however is not that useful in the HW/SW interface domain, since neither documentation nor the C-code are models.

Another definition of layers is used in the Eclipse Modeling-Framework (EMF) world (see Sect. 32.4.1.4). Here, the running program embedded in the Eclipse framework is seen as *world*, i.e., being level M0. Unfortunately the data of the program is nothing else than our model. Since they are part of the "world," a *model* in the Eclipse terminology defines the structure of the data, which is – in our terminology – a *metamodel*. Consequently, the terminology is shifted one level down and only has view, model, and metamodel.

For the rest of the book chapter, we will follow the widely used OMG definition as depicted in Fig. 32.8.

32.2.3 History and Known Technologies

Metamodeling is not as new as it sounds. The basic idea was introduced about 40 years ago by Chen. In those days, it was called entity-relationship model (see

[3]). Most of the metamodeling concepts introduced above were either already there or introduced some years later by Smith and Smith (see [29]). From there on, they were continuously improved, especially in the domain of databases. It is therefore no surprise that there is still a relationship between database schemas and metamodels. Entity relationship diagrams were also used in the definition phase of the Jessi Common Framework Initiative (see e.g., [35]), now used to define entities in the hardware domain. First, mainly structurally oriented entities were modeled. The definition of the *EDIF Information Model* by Kahn (see [18]) used the entity-relationship methodology also named information model here. Due to complexity, it was expressed in a textual form using the EXPRESS notation (see [26]). The entity-relationship notation was however not limited to structural things. Soon later Kahn and Guimale defined an information model for VHDL that covered behavior and time as well [14]. A summary of all the work around entity-relationship models of hardware was collected in [2]. Here, the term *metamodeling* was already used in conjunction with hardware modeling.

Unfortunately, research activities around metamodeling in the hardware domain cooled down for a while. The Open Access Database (see [15]), intended to store hardware design data, was, for example, associated with an API, however not with an unambiguous metamodel. Fortunately, metamodeling grew further in the software world under the umbrella of the OMG. The design of XML, UML, and other technologies was based on metamodels.

32.2.4 The Case for Metamodeling

After having seen that metamodeling is not that new a concept in hardware design, this section now discusses the benefits of metamodeling in the design process. We ended Sect. 32.2.1.2 with the statement that so far, metamodeling is just an additional view in the design of the HW/SW interface. The following carves out the benefits of metamodeling, making it a very useful technology in the TLM area and HW/SW interface automation area.

The first benefit is that the views illustrated in this chapter's examples – and many more – can be generated from a model. Therefore, generators have to be built that translate the content of the model to the syntax of the target view. So, all but one view – the model – can be completely derived from the model and no longer need to be developed manually.

The second benefit is that the content of the model must not necessarily be entered manually. Often, parts of the model are already defined in a specification. By providing a specification reader – or a single source for specification and model – additional time can be saved and consistency between specification and design views can be improved, if not guaranteed.

The third benefit is that big parts – especially the APIs – of the software needed to read the specification and to write the views can be automatically generated from a metamodel. Going hand in hand with that step, a good documentation via the metamodel diagram and a consistent way of treating models is achieved. This

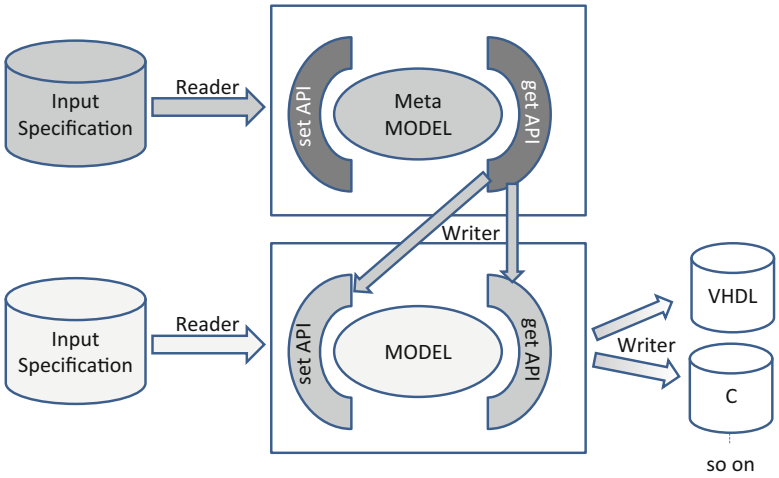


Fig. 32.9 Concept of a metamodeling framework

is nothing else than the application of modeling and generation techniques to the construction of the automation tools.

All this is summarized in Fig. 32.9. Since not stated explicitly yet, a further detail worth noting, the APIs needed to handle different metamodels can be also generated from the meta-metamodel.

In order to build such a framework in a safe way, a good formal basis is needed. Diagrams, as shown so far, give a good overview but are less suited to define all details. In the next part of this chapter, we therefore introduce the formal definition of models and metamodels.

32.3 A Formal Model of Metamodeling

This section provides a formalization of metamodels, their models and the relation between metamodel and model. The representation used in the following is based on a set-oriented perspective on models. This approach was selected as it permits an intuitive description of the constraints metamodels impose on their models.

32.3.1 Basic Definitions

Models consist of correlated objects which contain attributes. Both the correlations and the attributes are named. Moreover, the attributes are multisets that contain some values. The following definitions provide a formal definition of these names and of the values that can be stored in model attributes.

32.3.1.1 Legal Names *N* of Correlations and Attributes

All legal names in a model are grouped in the set N. In this formal model, we define this set as a set of words over an alphabet Σ. The characters selected for this alphabet can be picked from an arbitrary set. It is convenient to rely on a set of words that can be easily mapped to potential view languages (i.e., words that are legal identifiers in common programming and modeling languages). Equation 32.1 defines such a set N of valid names:

$$\begin{aligned} \Sigma_{alphabet} &= \{a, \ldots, z, A, \ldots, Z\},\ \Sigma_{numeric} = \{0, \ldots, 9\} \\ N &= \mathscr{L}(\Sigma_{alphabet}(\Sigma_{alphabet}|\Sigma_{numeric})^*) \end{aligned} \tag{32.1}$$

where $\mathscr{L}(e)$ describes the set of the regular language defined by the regular expression e.

32.3.1.2 Legal Values for Attributes

Attributes in models are typed. These types define the possible values an attribute can take and their interpretation. In this formal representation, type T_0 is defined by a set. This set contains all the values this type allows. An attribute of type T_0 can only take values $v \in T_0$.

In this formal model, some predefined types are provided. Each element in $T = \{I, B, F, S\}$ defines such a type. Each type $T_0 \in T$ is a set, containing all the values an attribute of the type T_0 may take.

The predefined sets have the following content:

- $S := \Sigma^* := \{w_0 \ldots w_n | n \in \mathbb{N}\} \cup \{\epsilon\}$ is the set defining the language of all valid strings. ϵ is the empty word and Σ is the alphabet of all possible characters.
- I is the set of all words representing valid integers.
- F is the set of all words representing valid floating point numbers.
- $B := \{True, False\}$ are the truth values of propositional logic.

The string type S occupies a special position here: It is a superset of any other type. Consequently, it contains any value an attribute – regardless of its type – may take:

$$\forall T_0 \in T : T_0 \subseteq S \tag{32.2}$$

Each of sets introduced above has an infinite number of elements. Metamodeling environments have predefined types such as integers of limited sizes and floating point numbers of various precision, originating from the runtimes used or the programming languages they are implemented in. They therefore provide subsets, constraining the range of integers, the precision of floating point numbers, and the length of elements in S which makes all type sets finite.

To work with attribute values, the metamodeling environment has to be able to interpret the elements in the type sets. For example, integers are used in the

metamodel to define the minimum and maximum multiplicities of elements in the model. To enforce these definitions and to work with the information they provide, the metamodeling environment has to understand the integer type: It has to provide a bijective mapping INT: $I \rightarrow \mathbb{Z} \cup \{\infty\}$. Using this and other mappings, a runtime environment can define operators such as addition and multiplication on integer or floating point numbers.

32.3.2 A Formal Representation of a Model

32.3.2.1 The Set of All Models

This formal representation of models puts emphasis on the fact that everything is a model. Section 32.2.2 illustrates that what is a model exclusively depends on the point of view on the modeled system. Any model A becomes a metamodel not by definition but because it is instance of a model we perceive as meta-metamodel. Likewise, a model A can become a metamodel if there is a model C that is an instance of A.

The set of all models M therefore contains models, metamodels, and the meta-metamodel. Every model $m \in M$ that is part of this set has the same structure which is defined in the following.

32.3.2.2 Definition of a Model

Each model $m \in M$ is a tuple

$$m = (n, mm, O) \tag{32.3}$$

where $n \in N$ is the name of the model (an element from the set of all identifiers). $mm \in M$ is the metamodel which m is instance of. The main part of the model is the set of objects O. It contains the actual elements of a model.

32.3.2.3 The Set of All Objects *O* of a Model

Each object $o \in O$ is a tuple:

$$o = (c, i, A, K, R) \tag{32.4}$$

where

- c is the class definition of the object. This definition is an object of the metamodel the model adheres to:

$$\forall m \in M,\ \forall o \in m.O : o.c \in m.mm.O \tag{32.5}$$

- $i \in I$ is the name or unique identifier of the object. As the name suggests, there are never two objects in a model which have the same identifier:

$$\forall o_i, o_j \in m.O : o_i.i = o_j.i \Leftrightarrow o_i = o_j \tag{32.6}$$

- A is a set of object specific attributes where each attribute $a \in A$ is a tuple of name and assigned values:

$$a = (n \in N, E), \quad E \subseteq S \tag{32.7}$$

  n is the name of the attribute and E is a multiset containing the attribute's values. If an attribute contains several identical values, the multiset E contains the value several times. For each attribute, there is only one tuple in an object's $o.A$ set. Equation 32.8 shows that if an attribute contains multiple values, these values are all in the multiset of the same attribute tuple.

$$\forall a_i, a_j \in o.A : a_i.n = a_j.n \Rightarrow a_i = a_j \tag{32.8}$$

The following will look at the elements K and R of the object tuple o. Analogous to the set of attributes A, both K and R are of a set of tuples $(n \in N, E)$, where n is a name and E is a set of values:

- K describes the children of the object o. Each element $k \in K$ is a tuple

$$k = (n \in N, E), \quad E \subseteq I \tag{32.9}$$

- R is the set of referenced objects. R describes referenced elements, a concept we did not yet introduce. They are still included here as they are an essential part of all modern metamodeling frameworks. References allow elements inside models to refer to other elements at object granularity. They can be compared to attributes containing a pointer to other objects instead of an attribute value. Each element $r_k \in R$ is a tuple

$$r = (n \in N, E), \quad E \subseteq I \tag{32.10}$$

Despite their largely similar structure, elements E which are part of tuples belonging to K and R have a different interpretation than elements belonging to tuples in A. The elements in E of an attribute contain actual values. In contrast, the elements E in a reference tuple r or child tuple k contain unique identifiers of other objects in the same model.

Example 1. The formal representation can now be used to describe any models as an element $m \in M$. Figure 32.10 provides an example for such a description. The model depicted here is the same model we used for our introductory example in Fig. 32.2.

Further Constraints Aside from the straightforward constraints already mentioned in the definition of the object tuple o, there are several further constraints:

- For every element $e \in E$, there is an object in the same model which has an identifier defined by the reference target t:

$$\forall m \in M,\ o_i \in m.O,\ r \in o_i.R, e \in r.E : \exists o_j \in m.O : o_j.i = e \tag{32.11}$$

$$
\begin{aligned}
m.O = \quad & \{o_{Simple}, o_{R0}, o_{R1}, o_{B1}, o_{B2}, o_{B3}, o_{B4}\} \\
o_{Simple} = \quad & (o_{Component}, 1, \{(n_{Name}, \{\texttt{Simple}\})\}, \\
& \quad \{(n_{Register}, \{2,3\})\}, \phi) \\
o_{R0} = \quad & (o_{Register}, 2, \{(n_{Name}, \{\texttt{R0}\}), (n_{Offset}, \{\texttt{0}\}), (n_{Size}, \{\texttt{16}\})\}, \\
& \quad \{(n_{Bitfields}, \{4\})\}, \phi) \\
o_{R1} = \quad & (o_{Register}, 3, \{(n_{Name}, \{\texttt{R0}\}), (n_{Offset}, \{\texttt{0}\}), (n_{Size}, \{\texttt{16}\})\}, \\
& \quad \{(n_{Bitfields}, \{5,6,7\})\}, \phi) \\
o_{B1} = \quad & (o_{Bitfield}, 4, \{(n_{Name}, \{\texttt{B1}\}), (n_{Offset}, \{\texttt{0}\}), \\
& \quad (n_{HWr}, \{\texttt{False}\}), (n_{HWw}, \{\texttt{True}\}), \\
& \quad (n_{SWr}, \{\texttt{True}\}), (n_{SWw}, \{\texttt{True}\}), \\
& \quad (n_{Size}, \{\texttt{16}\})\}, \phi, \phi) \\
o_{B2} = \quad & (o_{Bitfield}, 5, \{(n_{Name}, \{\texttt{B2}\}), (n_{Offset}, \{\texttt{0}\}), \\
& \quad (n_{HWr}, \{\texttt{True}\}), (n_{HWw}, \{\texttt{False}\}), \\
& \quad (n_{SWr}, \{\texttt{False}\}), (n_{SWw}, \{\texttt{True}\}), \\
& \quad (n_{Size}, \{\texttt{1}\})\}, \phi, \phi) \\
o_{B3} = \quad & (o_{Bitfield}, 6, \{(n_{Name}, \{\texttt{B3}\}), (n_{Offset}, \{\texttt{1}\}), \\
& \quad (n_{HWr}, \{\texttt{False}\}), (n_{HWw}, \{\texttt{True}\}), \\
& \quad (n_{SWr}, \{\texttt{True}\}), (n_{SWw}, \{\texttt{False}\}), \\
& \quad (n_{Size}, \{\texttt{1}\})\}, \phi, \phi) \\
o_{B4} = \quad & (o_{Bitfield}, 7, \{(n_{Name}, \{\texttt{B4}\}), (n_{Offset}, \{\texttt{2}\}), \\
& \quad (n_{HWr}, \{\texttt{False}\}), (n_{HWw}, \{\texttt{True}\}), \\
& \quad (n_{SWr}, \{\texttt{True}\}), (n_{SWw}, \{\texttt{False}\}), \\
& \quad (n_{Size}, \{\texttt{1}\})\}, \phi, \phi)
\end{aligned}
$$

Fig. 32.10 An example of a model in its formalized representation

Because of Eq. 32.6, there is exactly one element for every identifier and the reference is unique.

- Every object can be child of at most one element. In other words, every object has at most one parent. If we pick any two *different* objects o_i and o_j of the same model (therefore part of the same set $m.O$) and compare the intersection of their sets of children $o_i.K$ and $o_j.K$, we will find that it is the empty set. Equation 32.12 illustrates that this is true if and only if we look at two different objects o_i and o_j.

$$\forall o_i, o_j \in m.O \; \forall k_i \in o_i.K, k_j \in o_j.K : k_i.E \cap k_j.E \neq \phi \Leftrightarrow o_i = o_j \tag{32.12}$$

- Every object is child of exactly one other object except for one object which is said to be the model's root. Equation 32.13 shows that there is exactly one object for which no other object exists that contains the object in its set of children ($\exists!$ is the uniqueness quantification operator stating that *there is one and only one*)

$$\exists! o_i \in m.O \nexists o_j \in m.O,\ k \in o_j.K : o_i.i \in k.E \tag{32.13}$$

- The child relationship defines a hierarchy. For any two objects $o_i, o_j \in O$ where o_i is direct or indirect child of o_j, o_j must not be direct or indirect child of o_i. For any sequence of objects where each object is child of its predecessing object, the first object and the last object must not be equal:

$$\forall (o_0, \ldots, o_n) \in O^{n+1}\ \forall i \in [0, n-1] : \exists k \in o_i.K : o_{i+1}.i \in k.E \Rightarrow o_0 \neq o_n \tag{32.14}$$

- The constraint from Eq. 32.8 also applies to $o.K$ and $o.R$.

  In addition to that, the names n are not only unique within the attributes, children, and references of one object but also across the whole object. If an object has an attribute with the name n_0, there must not be any reference or child which has the same name. In other words, children, attributes, and references share the same namespace. The pairwise intersection of all child names, all attribute names, and all reference names must therefore be empty:

$$\forall o \in O\ \forall a, b \in o.K \cup o.A \cup o.R :\ a.n = b.n \Leftrightarrow a = b \tag{32.15}$$

Sets of All Models, Metamodels, and Meta-Metamodels This formal model defines three layers of models as illustrated in Fig. 32.8 of Sect. 32.2.2. These three layers are formalized here as three sets M, MM, and MMM. The largest of these sets is the set of all models M which was already introduced in the beginning of Sect. 32.3.2.

A metamodel $mm \in MM$ is a model that describes the structure of a model. A model $m_a \in M$ becomes a metamodel through the existence of a model $m_b \in M$ which is an instance of it. Metamodels are therefore also models and the set of metamodels is given by

$$mm = \{m \in M \mid \exists m_b \in M : m_b.mm = m\} \tag{32.16}$$

A model becomes a meta-metamodel through the existence of a metamodel which is instance of it. The set of meta-metamodels is therefore given by

$$MMM = \{m \in M \mid \exists m_b \in MM : m_b.mm = m\} \tag{32.17}$$

It is obvious that $MM \subset M$ and $MMM \subset M$. As any meta-metamodel necessarily has an instance, it is also a metamodel and the two equations simplify to $MMM \subset MM \subset M$:

$$\forall mmm_i \in MMM\ \exists m \in MM : m.mm = mmm_i \Rightarrow mmm_i \in MM \tag{32.18}$$

Based on this definition and the assumption that there is only one meta-metamodel, it is trivial that the meta-metamodel is its own metamodel:

$$\begin{aligned} \exists\, mmm_j \in MMM \mid mmm_i.m = mmm_j \\ |MMM| = 1 \Rightarrow mmm_j = mmm_i \end{aligned} \tag{32.19}$$

The meta-metamodel can therefore be used to describe itself. Equivalent notations for this are $mmm. = mmm$ or $mmm = (n, mmm, O)$

$$\forall mm_1, mm_2 \in MM : mm_1.mm = mm_2.mm = mmm \tag{32.20}$$

32.3.3 Metamodel Constraints on Models

In a bottom-up approach, the formal definition of models in formal models presented section 32.3.2.2 already lists some constraints on models that originate from the meta-metamodel we agreed on.

In the following, we complement this with other constraints a metamodel mm imposes on its models m. Approaching the problem top-down, we first declare names for attributes and objects of the meta-metamodel: $n_{name}, n_{ref}, n_{comp}, n_{attr}, n_{min}, n_{max}, n_{type} \in N$. Next, a formal description of the meta-metamodel mmm as instance of itself is developed. Finally, a list of constraints is provided.

Meta-Metamodel Any metamodel can then be uniquely represented as instance of this meta-metamodel mmm:

$$\begin{aligned} o_{class}, o_{composition}, o_{reference}, o_{attribute} &\in mmm.O \\ o_{class} &= (o_{class}, 1, A_{name}, K_{class}, \phi) \\ o_{composition} &= (o_{class}, 2, A_{name} \cup A_{multiplicity}, \phi, R_{class}) \\ o_{reference} &= (o_{class}, 3, A_{name} \cup A_{multiplicity}, \phi, R_{class}) \\ o_{attribute} &= (o_{class}, 4, A_{name} \cup A_{multiplicity} \cup A_{types}, \phi, \phi) \\ A_{name} &= \{(n_{name}, N)\} \\ A_{multiplicity} &= \{(n_{max}, I \cup \{\infty\}), (n_{min}, \underbrace{I}_{\text{Set of legal integer values } I \subset S})\} \\ A_{types} &= \{(n_{type}, \underbrace{S}_{\text{Set of all possible strings. Superset of any possible type-set } T_i.})\} \end{aligned} \tag{32.21}$$

$$\begin{aligned} K_{class} &= \{(n_{comp}, \{2\}), (n_{ref}, \{3\}), (n_{attr}, \{4\})\} \\ R_{class} &= \{(n_{type}, \{1\})\} \end{aligned}$$

Metamodels instantiate these uniquely named objects and provide values for their attributes. What we described above is in fact both the description of the meta-metamodel mmm and a metamodel mm (as it is instance of mmm).

We can now use the description of the meta-metamodel and provide formulae for how metamodel objects constrain their models. These descriptions are simplified by a set of helper functions.

Helper Functions The functions $O_{A,mm}$, $O_{K,mm}$, and $O_{R,mm}$ are defined for any model m. They map any object of the model to the set of metamodel objects that describe its attributes ($O_{A,mm}$), children ($O_{K,mm}$), and references ($O_{R,mm}$) or a combination thereof ($O_{KR,mm}$, $O_{KRA,mm}$). In programming, such a mapping is called introspection. Each of them therefore maps from the pre-image set $m.O$ of any model to the image set $\mathscr{P}(m.mm.O)$:

$$O_i \colon m.O \to \mathscr{P}(m.mm.O) \tag{32.22}$$

The function $\mathscr{P}(X) := \{U \mid U \subseteq X\}$ maps any set X to its powerset.

$$\begin{aligned}
O_{A,mm} &\colon o \mapsto \{o_i \in mm.O \mid \exists k \in o.c.K : o_i.i \in k.E \wedge k.n = n_{attr}\} \\
O_{K,mm} &\colon o \mapsto \{o_i \in mm.O \mid \exists k \in o.c.K : o_i.i \in k.E \wedge k.n = n_{comp}\} \\
O_{R,mm} &\colon o \mapsto \{o_i \in mm.O \mid \exists k \in o.c.K : o_i.i \in k.E \wedge k.n = n_{ref}\} \\
O_{KRA,mm} &\colon o \mapsto O_{A,mm}(o) \cup O_{K,mm}(o) \cup O_{R,mm}(o) \\
O_{KR,mm} &\colon o \mapsto O_{K,mm}(o) \cup O_{R,mm}(o)
\end{aligned} \tag{32.23}$$

Moreover, the function E_{name} returns the set of all values of an attribute provided the attribute name. The function o_{byid} returns an object given by its unique identifiers:

$$\begin{aligned}
E_{name} &\colon O \times N \to \mathscr{P}(S), \quad (o,n) \mapsto \{e \in S \mid \exists y \in o.A \cup o.R \cup o.K, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y.n = n : e \in y.E\} \\
o_{byid} &\colon M \times I \to O \qquad (m,i) \mapsto o \in m.O \mid o.i = i
\end{aligned} \tag{32.24}$$

Constraints The following constraints are valid for any model m of metamodel mm:

- For each attribute, composition, and reference, the metamodel restricts the minimum and maximum number of elements that are contained or referenced. This number is commonly referred to as multiplicity:

$$\begin{aligned}\forall y \in o.A \cup o.K \cup o.R,\ o_{mm} \in O_{KRA,mm}(o),\ e_{max} \in E_{name}(o_{mm}, n_{max}) :\\ y.n \in E_{name}(o_{mm}, n_{name}) \Rightarrow \text{INT}(e_{max}) \geq |y.E|\\ Y \in \{K, R, A\}\ \forall o_{mm} \in O_{Y,mm}(o),\ e_{min} \in E_{name}(o_{mm}, n_{min})\ \exists y \in o.Y :\\ y.n \in E_{name}(o_{mm}, n_{name}) \wedge \text{INT}(e_{min}) \leq |y.E|\end{aligned} \tag{32.25}$$

- For each attribute, the metamodel constrains the values that it can take. To do this, the metamodel object defining the attribute has a type property listing all possible values:

$$\begin{aligned}\forall a \in o.A,\ o_{mm} \in O_{A,mm}(o) : \quad & a.n \in E_{name}(o_{mm}, n_{name})\\ & \Rightarrow a.E \subseteq E_{name}(o_{mm}, n_{type})\end{aligned} \tag{32.26}$$

- Compositions and target references are statically typed through the metamodel. For each composition element and each reference, the metamodel constrains the metamodel object the that the target is instance of:

$$\begin{aligned}\forall m \in M,\ y \in m.o.K \cup m.o.R,\ i_{target} \in y.E,\ o_{mm} \in O_{KR,mm}(o) :\\ y.n \in E_{name}(o_{mm}, n_{name}) \Rightarrow o_{byid}(m, i_{target}).c.i \in E_{name}(o_{mm}, n_{type})\end{aligned} \tag{32.27}$$

32.4 Metamodeling for HW/SW Codesign

32.4.1 Metamodeling Frameworks

The formal models presented in Sect. 32.3 lay the foundation for the correct design and construction of metamodeling frameworks. A first framework has already been sketched in Sect. 32.2.4. This section first discusses Model-Driven Architecture (MDA) and its impact on metamodeling frameworks. It then describes the metamodeling frameworks most widely used for automated design creation: XML, UML, and EMF. Next, standardized metamodels covering the HW/SW interface are introduced and the use of custom metamodels is motivated. Practical experience gained from the use of metamodeling in industrial applications finishes this section.

32.4.1.1 MDA

MDA – acronym for Model-Driven Architecture – is a vision of the Object Management Group (OMG) (see [23] and [32]) for automation of code development via metamodeling, model-to-model transformations, and code generation. In the simple approaches toward metamodeling sketched in Fig. 32.9, a specification is first read into a model via a reader and then translated to the view via a generator. To

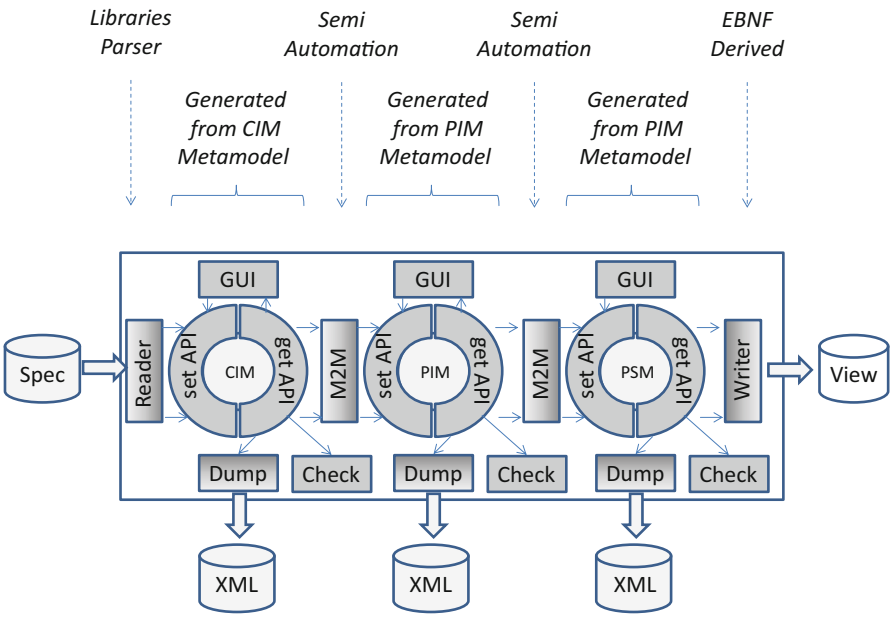


Fig. 32.11 An advanced metamodeling framework

implement the full MDA vision, this simple approach is extended: The target view is not generated directly from the first model, which is still close to the specification. Instead, the view is generated from a model that is closer to the target view's structure and semantics. Since these two models may still differ too much, one or even more intermediate models might be introduced to further partition the translation. Figure 32.11 shows such a process with one additional intermediate model.

To structure this approach and to find criteria for the definition of the models involved, MDA introduces three levels of models:

PSM: A Platform Specific Model is very close to the target view. Model and view therefore have similar structure but different syntax. In addition, the Platform Specific Model has sufficient information about the environment of the final view. A good example for a PSM is the Abstract Syntax Tree (AST) of the view's underlying language which already includes references to libraries that are used in the programming or modeling environment of the platform. For the example of the hardware register view, the PSM would be the AST of VHDL with links to VHDL's synthesis packages and libraries of reusable components.

If view generation starts from a PSM, the view can also be seen as an instance of the model and not just the content of the model since the view follows the structure of the model.

PIM: A Platform Independent Model does not include platform details but already depends on the targeted implementation's semantics – or in SW terminology: It depends on the kind of computation being targeted.

A good example is a view's language independent structural model being able to handle components, their instantiations and connections between them. However, such a model does not consider language specifics as VHDL's component-based instantiation, its port signals types (e.g., `std_logic`) and out semantics, or Verilog's wire-based style.

CIM: A Computation Independent Model focuses on items in the modeled domain, independent of their implementation. A very good example is the register model of our simple component, since it is independent of the target semantics (i.e., of a sequential programming language describing the SW access, of a TLM register model interfaced by a method, or of an event-driven VHDL model describing the HW part). This independence is precisely what is important for metamodels describing interfaces between domains. It is also easier to parse specification data into that model since a specification focuses on *what the design should do* and not on *how the design shall be implemented.*

In MDA, for each of the CIMs, PIMs, and PSMs, a metamodel is defined. This metamodel acts as the starting point and interface agreement for automation.

A metamodeling framework following the ideas of MDA is shown in Fig. 32.11. Here, the models are encapsulated in APIs with separate interfaces for writing and reading data (`setAPI` and `getAPI`). Using these interfaces, the data can be dumped into an intermediate storage based on the XML format, edited with a GUI or validated by check functions. These checks are derived from constraints provided by the metamodel or the meta-metamodel in use. All these components of metamodeling framework can be automatically generated using the metamodel description. Additional checks can be generated from OCL constraints associated with the metamodel or parts of it.

There are even several methods to automate the construction of translators. The view generator, for example, can either be derived from a metasyntax notation like EBNF or can be based on a template engine. The reader can, for example, make use of libraries or parser generators. All together, a metamodeling framework is a powerful tool that not only automates generation of views but also the construction of parts of the automation solution.

The metamodeling approach described in [8] and entitled *meta-synthesis* goes one step further. It also automates the building of automation tools by supporting merge and split of data during model-to-model transformation, dump and reload utilities for models, checks of model consistency, and execution control. Providing such a high level of automation makes it comprehensible how the use of models which are instances of metamodels helps to dramatically improve design productivity and design quality although the metamodels have to be built upfront.

In the following, we introduce the three metamodeling frameworks. EMF, synonym for Eclipse Modeling Framework, as well as XML and UML which describe both modeling languages and a modeling framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<Component Name="Simple">
  <Register Name="R0" Size="16" Offset="0">
    <Bitfield Name="B1" Size="16" Offset="0"
        HWr="false" HWw="true" SWr="true" SWw="true" />
  </Register>
  <Register Name="R1" Size="16" Offset="1">
    <Bitfield Name="B2" Size="1" Offset="0"
        HWr="true" HWw="false" SWr="false" SWw="true" />
    <Bitfield Name="B3" Size="1" Offset="1"
        HWr="false" HWw="true" SWr="true" SWw="false" />
    <Bitfield Name="B4" Size="1" Offset="2"
        HWr="false" HWw="true" SWr="true" SWw="false" />
  </Register>
</Component>
```

Fig. 32.12 The simple component model in XML format

32.4.1.2 XML

XML, acronym for eXtensible Markup Language, is a markup language that is used to store and annotate data. There are many books on XML and lots of web pages detailing the usage of the language in different fields of application. Instead of providing an overview over this vast area, this book focuses on the special features of XML that can be used in the context of metamodeling.

One of XML's initial goals was to separate content and view, an idea that also underlies the metamodeling concept. The mix of formatting and data – as, e.g., used in older versions of HTML – should be overcome to ease information retrieval and to support different publishing styles. The resulting XML standards were therefore shaped in a way that allowed storing data which could also be used to capture models of metamodeling environments. Figure 32.12 provides an example of our `simple` component model encoded in XML.

Of course, a representation similar to Fig. 32.12 can also be used to store metamodels. UML, for example, defines the XML Metadata Interchange (XMI) format as an XML-compatible markup language for storing models and metamodels. This XML Metadata Interchange (XMI) format is also used in the Eclipse EMF domain.

In addition, XML has a mechanism called XML Schema Definition (XSD) to define the valid structure of an XML document, effectively defining a metamodel for XML documents. XSD has a set of powerful features for specification of valid values. Similar to the formal definition in 32.3, XML and XSD only deal with strings. All values in the pictured XML file are therefore embedded in double quotes. An XSD schema specifies the valid strings an attribute may be assigned with and how those then have to be interpreted, e.g., as a number or a Boolean value. Figure 32.13 shows the XML schema defining the validity of the XML file in Fig. 32.12. It carries the same information as the metamodel that was introduced in the introductory example in Fig. 32.1. It is, however, not as intuitive as the graphical view of the metamodel. In addition, the requirements on well-formed XML documents make the format a bit verbose. XSD documents are formatted

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="component" type="Component" />
  <xsd:complexType name="Component">
    <xsd:sequence>
      <xsd:element name="register" type="Register" />
    </xsd:sequence>
    <xsd:attribute name="Name" type="xsd:string" use="required" />
  </xsd:complexType>
  <xsd:complexType name="Register">
    <xsd:sequence>
      <xsd:element name="bitfield" type="Bitfield" />
    </xsd:sequence>
    <xsd:attribute name="Name" type="xsd:string" use="required" />
    <xsd:attribute name="Offset" type="xsd:int" use="required" />
    <xsd:attribute name="Size" type="xsd:int" use="required" />
  </xsd:complexType>
  <xsd:complexType name="Bitfield" minOccurs="1" maxOccurs="unbounded">
    <xsd:attribute name="Name" type="xsd:string" use="required" />
    <xsd:attribute name="Offset" type="xsd:int" use="required" />
    <xsd:attribute name="Size" type="xsd:int" use="required" />
    <xsd:attribute name="HWr" type="xsd:bool" use="required" />
    <xsd:attribute name="HWw" type="xsd:bool" use="required" />
    <xsd:attribute name="SWr" type="xsd:bool" use="required" />
    <xsd:attribute name="SWw" type="xsd:bool" use="required" />
  </xsd:complexType>
</xsd:schema>
```

Fig. 32.13 Component metamodel in XSD format

in valid XML themselves. This allows existing XML parsers to read the schemas; however, it is also responsible for XSD's verbosity.

Specific to the XML-based MDA process is that the model is stored in a file and not in an encapsulated data model that is part of a program. This is shown in Fig. 32.14 on the same MDA flow as pictured in Fig. 32.11. Here, the transformations are done by XML processors which read XSLT, a transformation language for XML.

XSLT can also be used to translate XML to any kind of textual view such as C-code or VHDL models. The only thing needed to build an XML CIM in the depicted XML MDA flow is therefore a reader. If the specification is available as an XML document, this reader might be an XSLT-based translation as well.

32.4.1.3 UML

UML is a widely used standard in the software world incorporating many concepts and notations that have proven to be successful. UML also defines an aligned graphical view on all the concepts included. The widespread adoption of UML brought a significant benefit compared to the situation before where different methodologies used different graphical notations for the same concepts or even the same notation for different concepts. UML as a whole however has a disadvantage: It is quite complex and some definitions are ambiguous.

All the notations – also called diagrams – are defined via a metamodel which in turn is defined on basis of a meta-metamodel. This UML meta-metamodel is called MOF (see [22]) – acronym for Meta Object Facility. Inside the MOF and essential MOF called EMOF and a complete MOF called CMOF is defined. Based on MOF, UML defines a standard intermediate called XMI which we already mentioned.

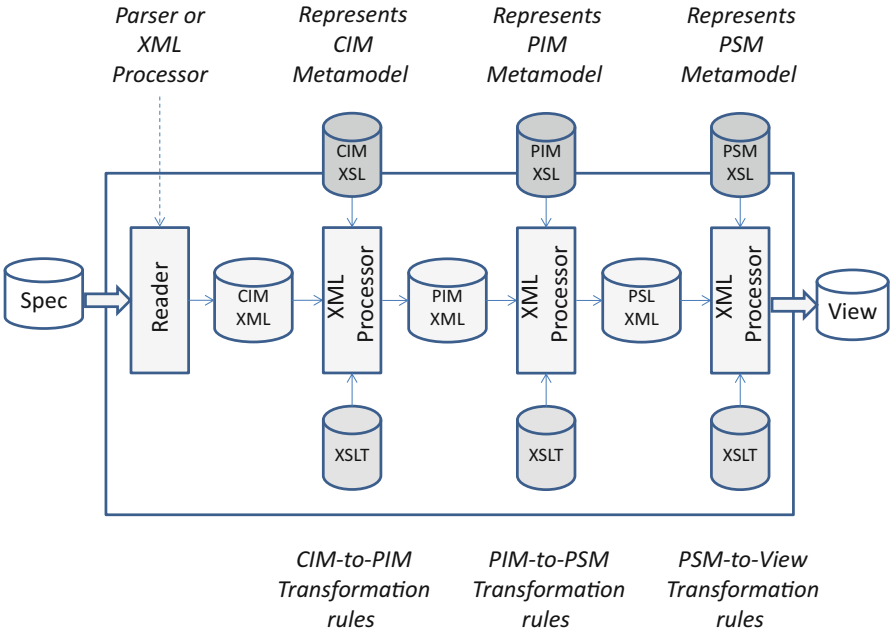


Fig. 32.14 MDA using XML

UML influenced metamodeling in many ways. Since UML diagrams also support behavioral notations like state or activity diagrams, it proves that metamodels are not restricted to structural information. Instead, a structure can be defined which has inherent execution semantics.

UML also supports class diagrams, which – or more precisely an extended subset of which – can be used to define metamodels in a graphical way. UML's EMOF is thus conceptually very close to a class diagram. This is obvious since a class diagram structures data and a metamodel structures its domain in entities (*classes*), their properties (*their attributes*), and their relation to other entities (*different kinds of associations*).

Last but not least, UML has a built-in extension mechanism, which permits the adaption of UML to different needs and domains. The adaptability to different needs in particular is a noteworthy benefit of a modeling-based approach over the tool-based approach in system-level automation.

Stereotypes can be used to create new model elements, represented by «. . . »-brackets. These model elements may also have their own graphical representation. In addition, stereotypes can be used to create completely new diagrams including their graphical representation.

The second mechanism is *tagged values*. They used to define additional properties for existing modeling elements or stereotypes. They include additional information needed for specific use cases. This information can also influence further processing of the model. When our simple component metamodel is used

for HW generation, it can be extended with tagged values describing how many flip-flop instances should be generated for every bitfield:

- Zero: Only wires would be generated in both directions.
- One: Only one flip-flop group written by SW (as depicted in Fig. 32.3)
- One shared: Flip-flops writable by HW and SW
- Two: One flip-flop group being written by SW and read by HW as well as one flip-flop group being written by HW and read by SW

Other tagged values might specify if SW read and write accesses trigger pulses and edges at additional signals passed to the HW core.

It is important to note that tagged values modify the metamodel in an upward compatible way since they only add items. In contrast, UML's third adoption mechanism called *constraints* adds new rules or modifies existing ones. Constraints can be used to remove an attribute from a modeling element. Code generators that rely on a removed field would fail.

A set of any of the defined extensions can be packed and provided as a so-called profile. SysML (see Sect. 32.4.2.2) is one example of such a profile. These profiles are called lightweight extensions since they don't change UML's metamodel. In contrast, a heavyweight extension would add new items, concepts, and relationships to UML's MOF.

32.4.1.4 Eclipse Modeling Framework

EMF, synonym for Eclipse Modeling Framework, is a full-featured metamodeling framework. EMF uses Java as implementation and glue language and generates a Java API as well as other things for the specified metamodel (e.g., an editor model). EMF is fully integrated in the Eclipse framework. EMF is a good starting point for metamodeling since it is open source and can easily be obtained from the Eclipse Foundation. EMF's web page (see [9]) refers to many online tutorials, webcasts, and video-casts. For those preferring old-style printed books, the key contributors to EMF collected an overview on EMF in [30]. There are also many forums around EMF helping with questions.

Diving into EMF is however not straightforward and it takes a while to get started with the framework. Many consulting companies around EMF offer their help. If someone prefers closed-source tools that come bundled with professional service, then he/she might look at the tools from MetaCase [20]. An important contribution of EMF to the metamodeling world is its meta-metamodel called ECORE. ECORE was designed to be able to map XML schema, UML metamodels (diagram types in UML terminology), and database schemas to one model. The ECORE model shown in Fig. 32.15 is conceptually identical to our first meta-metamodel shown in Fig. 32.6.

Figure 32.15 shows four additional modeling features: First, as already included in our formal model, ECORE has a reference mechanism implemented via `EReference`. This mechanism has similarities with association in EMOF. Via `containment`, references and compositions (the only hierarchical element in our

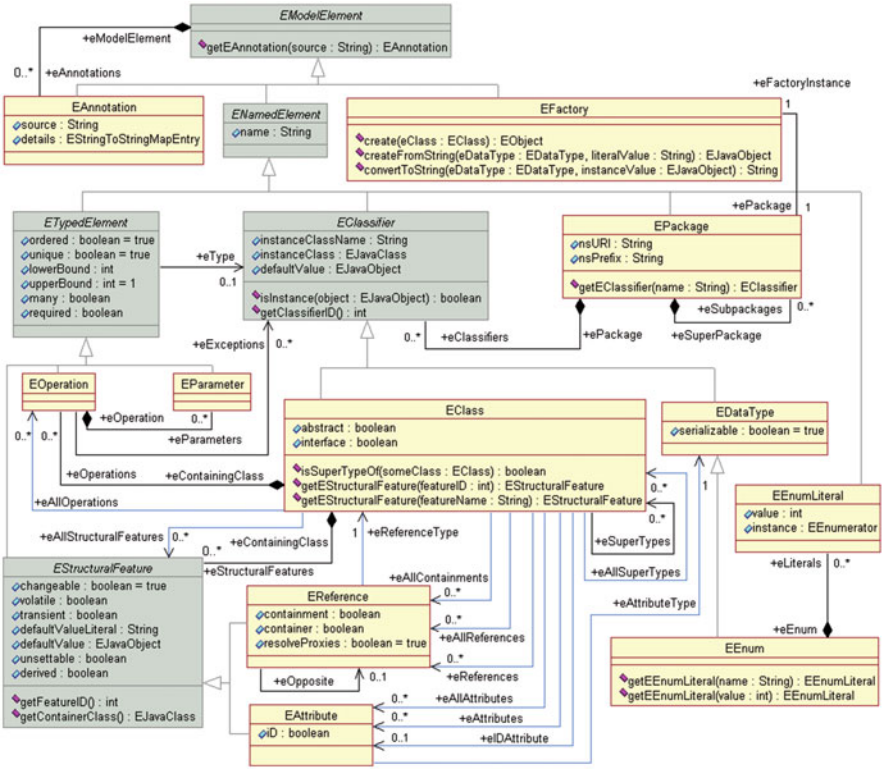


Fig. 32.15 EMF ECORE meta-metamodel [10]

simple metamodel) can be distinguished. Second, ECORE supports inheritance via `eSuperTypes` and `eAllSuperTypes`. Third, ECORE supports namespaces via `EPackage`. Fourth and last, ECORE supports enumerations via the `EEnum` and `EEnumLiteral` meta-classes.

Further, ECORE makes more use of inheritance. For example, all the naming is defined in the virtual classes `EClassifier`, `ENamedElement`, and `EModelElement`. Similarly, bounds and other features are derived from `EStructrualFeature` and `ETypedElement`.

The meta-class `EFactory` does not describe a modeling feature but methods to create the instances and to do string conversion. Finally, `EAnnotation` provides a measure to add data to the model that can be used, e.g., for view generation or model transformation.

Less obvious, yet just as important is that the API does not only permit access to the model. Instead, it also permits access to the metamodel items which are associated with the model items. In this way, introspection is supported for all model elements and meta-programming techniques can be applied. This allows different attributes to be handled by the same piece of code although they are differently

typed. This facilitates the implementation of translators and view generators that support on any instance of one kind of meta-metamodel.

32.4.2 Related Standards

Around the XML metamodeling technology, two standards have been defined that are widely used in the HW/SW area: IP-XACT and SysML. This section gives an overview over the standards and afterward motivates the benefit of an application specific metamodeling approach.

32.4.2.1 IP-XACT

IP-XACT (see [19]) is a standard supporting automation of IP Integration and thereby automation of System-on-Chip (SoC) construction. A PDF version of the standard is available from the IEEE [16]. IP-XACT has wide professional support. Several Electronic Design Automation (EDA) tools support IP-XACT and almost all IPs have an associated IP-XACT view. There are also open-source tools supporting IP-XACT, e.g., Kactus2 (see [31]).

IP-XACT defines an XML Schema with additional semantic documentation of the schema items. From the modeling standpoint, IP-XACT primarily supports the definition of the following items:

- Signals, interfaces, and bus structures as elements for the connection of components.
- RTL, TLM, or mixed RTL and TLM connections.
- Components describing hardware blocks of the IP. To interface with the HW world components offer interfaces for complex signal bundles and simple ports. Further, they can have parameters. Finally, components include the definition of the register layout and thus define most parts of the IP's SW interface.
- Definition of connection of IPs in a so-called System Model.

From HW/SW perspective, only a subset of the features IP-XACT provides are of interest.

The System Model, since it defines the involved components, the number how often they are instantiated and the instance names. The definition of the base address for each instance and derived from that the based addresses of the register fields is an important key for efficient software development.

The way how registers are specified is more advanced than in our simple model. Components have addressable units specifying their own base address inside the address space of the component and their own address range. Furthermore, the addressable units can be connected to interfaces. Like this, registers can be addressed via two or more CPU buses or over other protocols such as Serial Peripheral Interface (SPI). Figure 32.16 illustrates that the registers in IP-XACT and our register model share a similar underlying concept: Components have registers and registers have bitfields.

```
<ipxact:register>
   <ipxact:name>R0</spirit:name>
    <ipxact:addressOffset> 0x00 </ipxact:addressOffset>
    <ipxact:size>16</ipxact:size>
    <ipxact:field>
       <ipxact:name>B1</ipxact:name>
       <ipxact:bitOffset>0</ipxact:bitOffset>
       <ipxact:bitWidth>16</ipxact:bitWidth>
       <ipxact:access>read-write</ipxact:access>
       <ipxact:volatile>true</ipxact:volatile>
    </ipxact:field>
</ipxact:register>
<ipxact:register>
   <ipxact:name>R1</spirit:name>
    <ipxact:addressOffset> 0x01 </ipxact:addressOffset>
    <ipxact:size>16</ipxact:size>
    <ipxact:field>
       <ipxact:name>B2</ipxact:name>
       <ipxact:bitOffset>0</ipxact:bitOffset>
       <ipxact:bitWidth>1</ipxact:bitWidth>
       <ipxact:access>write-only</ipxact:access>
       <ipxact:volatile>false</ipxact:volatile>
    </ipxact:field>
    <ipxact:field>
       <ipxact:name>B3</ipxact:name>
       <ipxact:bitOffset>1</ipxact:bitOffset>
       <ipxact:bitWidth>1</ipxact:bitWidth>
       <ipxact:access>read-only</ipxact:access>
       <ipxact:volatile>true</ipxact:volatile>
    </spirit:field>
       <ipxact:name>B4</ipxact:name>
       <ipxact:bitOffset>2</ipxact:bitOffset>
       <ipxact:bitWidth>1</ipxact:bitWidth>
       <ipxact:access>read-only</ipxact:access>
       <ipxact:volatile>true</ipxact:volatile>
    </spirit:field>
</ipxact:register>
```

Fig. 32.16 IP-XACT code fragment of registers

But IP-XACT also has some nice additional capabilities. For example, read and write access can be specified for address fields, registers, and bitfields. Further, the access is not defined via read and write flags but via `access`-field that can take the values `read-only`, `write-only`, as well as `read-write`. Also, `writeOnce` and `read-writeOnce` are supported.

In addition, IP-XACT allows to specify legal values for the bitfields. Each of the value items has a name, a value, and a description. In software, they can be mapped to enumeration types or macros making the SW access more readable.

Further, accesses can be byte accesses and bridge different types of endianness. Therefore `ipxact:endianness` and `ipxact:addressUnitBits` are associated with buses and address fields but not to singe registers.

Another important thing is the possibility to define the display name of registers and bitfields in addition to name and description. This display name may be used in the firmware headers since it can be easier to map it to target SW languages.

There is also a possibility to specify dimensions for registers, which allow – together with address fields – the hierarchical specification of the software-side

register interface in a more structured way based on `struct`- and `array`-constructs.

However, IP-XACT focuses on IP integration and therefore describes only the SW side of the component. The hardware side is only covered by the attribute `volatile`. This attribute tells if the hardware may change the value written by the software. The way how and where the value is stored and how the HW access is done cannot be specified in IP-XACT.

32.4.2.2 UML/SysML

This section focuses on SysML, an extended subset of UML, and describes where SysML modifies, adds, and skips UML diagrams or their features. Although both UML and SysML only speak of *diagrams*, these diagrams implicitly rely on an underlying metamodel, which all valid diagrams have to adhere to. This metamodel also provides semantics for the diagrams. There are many commercial and open-source SysML and UML tools available. In the Eclipse domain, the plugin Papyrus is widely used (see [11]).

UML and SysML diagrams can be subdivided into structural and behavioral diagrams. To describe behavior, UML and SysML introduce the notion of actions as basic items for functionality. SysML's behavior diagrams are:

- *Activity Diagrams*: These diagrams are a bit different in SysML and UML. They consist of activities that specify transformations of inputs to outputs and actions responsible for the transformation. Activities produce and consume artifacts that might be passed via flow ports. Based on an underlying semantic of colored Petri nets, activities may have control and data flow inputs and outputs. Both can trigger the execution of activities. Activity diagrams support the specification of hierarchies as well.

  Activities may be mapped to HW, SW, or mixed activities. In this case, artifacts are data being transferred between HW and HW, SW and SW, and HW and SW. Activity diagrams are therefore a measure to specify HW/SW partition.
- *State-Machine Diagrams*. They are the classical hierarchical (and parallel) program state machines and are the same in UML and SysML. State machines are primarily used for the specification of either HW or SW. Their hierarchy is used to structure complex descriptions and to enable parallelism. Change of states via transitions can be triggered by events. These events can be change, time, or signal events but they cannot be flow driven.
- *Sequence- and Use-Case Diagrams*: They specify single scenarios outside or inside a component (which may be, e.g., an activity or a block). These scenarios show interactions between items and have timing and branch features. Sequence- and use-case diagrams are also the same in UML and SysML.

The second group of diagrams are structural diagrams whose most important representatives are block diagrams and package diagrams.

- *Block Diagrams*: SysML distinguishes block definition diagrams and internal block diagrams. Both differ from their UML counterpart.

  Blocks are basic structural elements like hardware and software and therefore also used to define the HW/SW partitioning. Block definition diagrams visualize the external connection and internal block diagrams – as the name says – internal connections. Blocks have among others attributes and constraints. The block and its items can be interlinked with functions implementing the block and requirements to be fulfilled by the block.

  Blocks have standard UML ports which describe the classical provides/requires semantic. Blocks also have flow ports describing a data flow to and from the component.
- *Package Diagrams*: They are the same in UML and SysML. Package diagrams group model elements to a namespace which can also be used, e.g., for visualization in the tree browser. Packages also support views and viewpoints to group model elements from different packages by their relevance for a specific stakeholder.

In addition, SysML has a third diagram type called *Requirement Diagram*. Here, requirements can be interlinked with model items. Aside from a top-down interlinking of specification and design items to requirements, this helps to analyze the requirements by mapping to other model items.

Similar to IP-XACT, SysML has a lot of additional features that make the diagrams more usable. From the HW/SW perspective, it is worth mentioning that SysML also supports allocation of items to blocks, which are then called resources. The mapping of SW to specific processing elements but also the overall implementation of single blocks in hardware and software can be specified in this way. From UML's perspective, SysML lacks structural diagrams, object and class diagrams, as well as behavioral and timing diagrams.

Essential for both UML and SysML is the profile diagram, which does not provide new modeling features but the possibility to define new diagram types or to extend and constrain existing diagrams further. This is important since specific design challenges need additional information or additional capabilities for code generation – despite the huge number of predefined UML features. This holds true for IP-XACT as well.

32.4.2.3 Application Specific Metamodels

Sections 32.2 and 32.3 introduced metamodeling as a generic technology for automation of tool development, which in turn automate the generation of views in the HW/SW domain. The previous subsection describes standardized metamodels; it however also mentioned that despite the richness of the existing metamodels, they are not sufficient to cover wide ranges of the tasks necessary for system-level automation.

The Need for Design-Task Specific Metamodels

Let's recap why the existing point tools cannot cover the system-level automation area: System level is too heterogeneous and too complex to be covered by one

or a small number of tools. Not mentioned there, but clearly understandable, is that system automation tools operate in the area of concept engineering, overall functionality, and architecture design and therefore have to target all ways and styles an SoC can be designed in. In addition to the domain-specific issues, system-level automation tools have to support many things specific to companies or even to individual design groups to be able to widely automate the design process.

Moreover, design challenges steadily increase. Among others, power control and energy management, reprogrammable architectures, reliability and safety issues of modules, or access control must be supported. To make things worse, all these things have different goals and measures in different domains. Due to the *More Moore* and *More than Moore* trends, new automation will have to be supported, both in new application domains and in new design techniques.

Does this mean that the metamodel standards are entirely useless? Definitely not, since both UML and SysML provide extension mechanisms. For example, IP-XACT allows putting additional data at specific places in the model. These places are called `vendor extensions` as they are intended for IP Vendors and IP-XACT tool providers. Since the range of these vendor extensions is a bit limited, a metamodel designer does not have to exclusively rely on these places and can simply extend the IP-XACT schema according to his/her needs and make use of the existing concept in there. A simple filter with XSLT can remove the extensions and a legal IP-XACT model can be derived whenever required. The availability of XML-technology as open-source software, in many flavors and for many programming languages, makes it easy to write custom code generators .

SysML and UML with their built-in profile mechanisms are even more powerful in terms of extensions. Since the original metamodels do not need to be changed, there is no need to do backward transformations for all the extensions.

To sum up, predefined metamodels do not render to construction of custom metamodels – precisely tailored to ones need – unnecessary. Instead, they offer a good starting point and simplify making metamodels fitting to domain-specific models.

Utilization of Metamodels

Many very positive results in using metamodeling in SW design were reported. Metamodeling approaches however also gain momentum in the hardware world and thus in the SoC world covering hardware and software. The amount of research work in that field increases and a growing number of companies show interest in metamodeling.

At Infineon, a metamodeling framework was developed on the basis of Python. To simplify view generation, the Mako template engine is used. The modeling capabilities are about as powerful as in EMF, but no introspection layer is needed, since Python is rich in introspection capabilities from scratch.

The Infineon framework supports the classical levels from meta-metamodeling to view generation. Meta-metamodeling is also used to generate metamodels, i.e., to further increase the productivity when using the technology. Furthermore,

several utilities such as model comparison or GUI generation are based on meta-metamodels and thus provide automation for all designed metamodels.

More important, there are about 100 metamodels at the moment with even more generators in use, covering all modeling aspects mentioned above – those supported by standard metamodels and those not supported.

Benefits are formidable. Up to 60% effort reduction in implementing one chip or over 95% effort reduction in selected design steps speak for itself. Continuously, new metamodels are developed to further automate design.

32.4.2.4 A Peek into the Future of Metamodeling

Will metamodeling arrive in major EDA companies and subsequently as EDA metamodeling frameworks at their customers? This is not inconceivable as EDA companies already provide old-school imperative Tcl and other scripting interfaces to their tools. However, the following three blocking points need to be resolved:

First, metamodeling requires good object-oriented modeling skills, which is not a basic skill of every HW designer. The foundations for this have however been laid: Object orientation is a basic concept of widely used languages such as SystemC or SystemVerilog and related modeling techniques are gradually becoming a more important part of higher-level education.

Second, metamodeling eases building and adopting tools, and metamodels associated with intermediate formats help designers to do a lot of further automation. Unfortunately, the isolation strategy of the big EDA companies prevents them from properly supporting metamodeling. There are however small EDA companies that provide building blocks such as HDLs parsers which are easily linkable with metamodeling frameworks.

Third and last, metamodeling links design teams, concept engineers, and customers much closer together. An integrated and aligned design flow is needed to closely synchronize their activities. This is a matter of design culture and may take longest.

It is clear that due to the high availability of the technology and due to the growing amount of experience in its utilization, metamodeling will play a dominant role in system-level automation. It is however not clear who will provide metamodeling solutions to users: the big EDA companies or consulting companies building their business model around open-source technologies.

32.5 Generation

So far, this book chapter gives an overview on metamodeling techniques around the HW/SW interface including meta-metamodels, standard and specific metamodels, and the abstraction inherent in models. It also mentions, yet does not cover in detail, the aspect of generation, which we address in the concluding part of this chapter.

Early interest in generation was driven by VHDL-based reuse activities and utilizing the generate statement for generation of component alternatives in hardware

design (see, e.g., [25]). Similar approaches, using generative approaches built in languages, have been followed in SW domain and have been named *generative programming* (see, e.g., [4]). An enhanced preprocessor [5] even supporting iterative directives showed some improvements in coding productivity but didn't result in a breakthrough. Even though several more approaches have been made, generation has a shadowy existence in hardware design. Main reasons are the insufficient features of the built-in generation constructs, the verification challenge of the parameter specs, and the complexity setting the configuration parameters correctly.

Two trends carried generation forward. First, the introduction of generator chains in IP-XACT. They include treating the generator as a design view and support component parameter and parameter propagation. The benefit is that any notation, e.g., script language or advanced programming language, can be used to build the generator. The limitation of built-in generator constructs is solved in this way. Also, the generation of test benches have been introduced so that each parameter setting could be verified automatically on demand.

Second, the introspection capabilities of programming languages increased. Examples are Python, Scala, or the mentioned Java-based introspection in EMF. One prominent representative in the hardware domain is the work on Chisel [1]. It proposes to use Scala to define a hardware description language with generation capabilities based on introspection. This enables the generation of various design views such as RTL or functional models.

Besides advances in technology, the demand for generation increases since upfront estimations of the impact of architecture and component alternatives are hard if even not impossible. *Rethinking Digital Design* [28] thus proposes the use of generators as essential technology for future SoC designs.

Where these approaches share with metamodeling the idea of a need for flexibility in design and generation of design views, interface to specification and specification of interfaces, e.g., between HW and SW is not that well covered.

In this area, the idea of modeling entities, their attributes and associations, using metamodels is simply more powerful. Since metamodels provide the possibility for graphical entry, their usage and documentation is easier than that of their language-based counterparts.

Also, any kind and strategy for target view generation can be applied in metamodeling context. Popular strategies include:

- Simple writers as already mentioned in conjunction with IP-XACT generator chains. These writers may be generated from a concrete or Abstract Syntax Tree. Conformance to target languages is therefore guaranteed.
- Model-to-model translators as part of a Model-Driven Architecture approach. Including them helps to partition the complexity of generators in a systematic way.
- Template engines that allow to enter target code and extend it with generation pragmas step-by-step. The benefit is that a mix of code typing and generation can be easily established. Focus on these parts of the code that can take benefit from generation can be easily achieved.

To sum up, generation is the missing piece in the jigsaw of metamodeling. In addition to early structuring and analysis that can be done with metamodeling alone, generation provides better consistency between specification and design and between different but related design views – which is particularly important for the HW/SW interface. Furthermore, generators provide better code quality which helps reducing the number of bugs. Aside from the automation of typing, code generation can help avoid debugging efforts which is an additional pillar in productivity increase.

32.6 Conclusion

"Meta" – analogously translated as "beyond" – describes an abstraction by definition of the structure of the related view. A metamodel thus defines the structure of a model and a metametamodel defines the structure of a metamodel. This means, a metametamodel is the metamodel of a metamodel. Metamodels can be formally defined giving them a sound theoretical foundation.

Metamodeling techniques are known for over a quarter of a century reflecting their use in database schemas, XML schemas or the EXPRESS language. The Eclipse Modeling Framework (EMF) is an open source option to make own metamodels and build metamodel based applications.

The two major predefined in the HW/SW domain are IP-XACT and UML/SysML. IP-XACT defines, inter alia, registers and their bit-fields, i.e., the physical layer of the HW/SW interface. UML/SysML has graphical formalisms for the definition of behavior and structure, e.g., state diagrams or activity diagrams such as behavior diagrams and class diagrams or component diagrams.

Key for the productivity increase gained by metamodeling techniques is code generation, either by code generation of views from models with code generators conforming to the model's metamodel or by code generation of parts of a metamodeling framework with code generators conforming to the metamodel's metametamodel.

Applying metamodeling to the HW/SW interface allows saving of up to 95% of the design effort by generating various styles of documentation views, TLM-views (or other views beyond RTL), RTL-views, firmware views and verification views. UML/SysML expand the generation scope; however each of them is used only in either the hardware or the software domain.

Although metamodeling is very successfully used in the HW/SW domain, there are a lot of further opportunities to use metamodeling techniques in this domain, e.g., for the generation of low level drivers.

References

1. Bachrach J, Vo H, Richards B, Lee Y, Waterman A, Avizienis R, Wawrzynek J, Asanovic K (2012) Chisel: constructing hardware in a scala embedded language. In: DAC, pp 1216–1225

2. Bergé JM, Levia O, Rouillard J (eds) (1996) Meta-modeling – performance and information modeling. Kluwer Academic Publishers, Dordrecht
3. shan Chen PP (1976) The entity-relationship model: toward a unified view of data. ACM Trans Database Syst 1:9–36
4. Czarnecki K, Eisenecker U (eds) (2000) Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York
5. Ecker W (1998) Generative structural modeling using a VHDL pre-processor. In: Proceedings of the forum on design languages (FDL), Lausanne
6. Ecker W, Müller W, Domer R (eds) (2009) Hardware-dependent software: principles and practice. Springer, Berlin. http://opac.inria.fr/record=b1129256
7. Ecker W, Velten M, Zafari L, Goyal A (2014) The metamodeling approach to system level synthesis. In: Fettweis G, Nebel W (eds) DATE. European Design and Automation Association, pp 1–2
8. Ecker W, Velten M, Zafari L, Goyal A (2014) Metasynthesis for designing automotive SoCs. In: DAC. ACM, p 6. doi:10.1145/2593069.2602974
9. Eclipse Foundation (2016) Eclipse modeling framework. https://eclipse.org/modeling/emf
10. Eclipse Foundation (2016) ECORE relations. http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/doc-files/EcoreRelations.gif
11. Eclipse Foundation (2016) Papyrus – modeling environment. https://eclipse.org/papyrus
12. Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y (2003) Taming heterogeneity – the ptolemy approach. In: Proceedings of the IEEE, pp 127–144
13. Geisler NL (ed) (1999) Baker encyclopedia of Christian apologetics. Baker Books, Grand Rapids
14. Guimale C, Kahn H (1995) Information models of VHDL. In: Proceedings of the 32nd annual ACM/IEEE design automation conference
15. Guiney M, Leavitt E (2006) An introduction to openaccess: an open source data model and API for ic design. In: Proceedings of the 2006 Asia and South Pacific design automation conference. IEEE Press, pp 434–436
16. IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows. doi:10.1109/ieeestd.2014.6898803. http://standards.ieee.org/getieee/1685/download/1685-2014.pdf
17. Infineon Technologies (2014) XMC1100 AA-step reference manual. http://www.infineon.com/dgdl/Infineon-xmc1100-AA_rm-UM-v01_01-EN.pdf?fileId=db3a30433cfb5caa013d1600856033eb
18. Kahn H (ed) (1995) EDIF version 350/400 and information modelling. EDIF Technical Centre, Manchester University
19. Kruijtzer W, Van der Wolf P, De Kock E, Stuyt J, Ecker W, Mayer A, Hustin S, Amerijckx C, De Paoli S, Vaumorin E (2008) Industrial IP integration flows based on IP-XACT standards. In: Design, automation and test in Europe (DATE 2008). IEEE, pp 32–37
20. MetaCase (2016) MetaCase. https://www.metacase.com/de/
21. Nikolic B (2015) Simpler, more efficient design. In: ESSCIRC conference 2015 – 41st European solid-state circuits conference, Graz, 14–18 Sept 2015, pp 20–25. doi:10.1109/ESSCIRC.2015.7313819
22. Object Management Group (2015) OMG meta object facility (MOF) core specification. http://www.omg.org/spec/MOF/2.5
23. OMG (2016) MDA – the architecture of choice for a changing world. http://www.omg.org/mda
24. Pagès B, de Freitas LG, Yeşilyurt H (2016) DoUML. https://github.com/DoUML/douml
25. Preis V, Henftling R, Schütz M, März-Rössel S (1995) A reuse scenario for the VHDL-based hardware design flow. In: Proceedings EURO-DAC'95, European design automation conference with EURO-VHDL, Brighton, 18–22 Sept 1995, pp 464–469. doi:10.1109/EURDAC.1995.527445
26. Schenck DA, Wilson PR (eds) (1993) Information modeling the EXPRESS way. Oxford University Press, New York

27. Schneider C (1997) A parallel/serial trade-off methodology for look-up table based decoders. In: DAC, pp 498–503. doi:10.1145/266021.266213
28. Shacham O, Azizi O, Wachs M, Richardson S, Horowitz M (2010) Rethinking digital design: why design must change. IEEE Micro 30(6):9–24
29. Smith J, Smith D (1977) Database abstractions: aggregation. ACM Trans Database Syst (TODS) 2(2):105–133
30. Steinberg D, Budinsky F, Paternostro M, Merks E (eds) (2008) EMF: eclipse modeling framework. Pearson Education
31. Tampere University of Technology (2016) Kactus2. http://funbase.cs.tut.fi/
32. Truyen F (2006) The fast guide to model driven architecture. http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf
33. Warmer J, Kleppe A (eds) (2003) The object constraint language: getting your models ready for MDA 2. Addison-Wesley Longman Publishing Co., Inc., Boston
34. Wikipedia (2016) Backus-naur form – Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Backus-Naur_Form?oldid=717943564#Further_examples
35. van der Wolf P (1994) Cad frameworks: principles and architecture. The Springer international series in engineering and computer science. Springer, Boston. https://books.google.de/books?id=R4BRAAAAMAAJ

Hardware/Software Codesign Across Many Cadence Technologies

33

Grant Martin, Frank Schirrmeister, and Yosinori Watanabe

Abstract

Cadence offers many technologies and methodologies for hardware/software codesign of advanced electronic and software systems. This chapter outlines many of these technologies and provides a brief overview of their key use models and methodologies. These include advanced verification, prototyping – both virtual and real, emulation, high-level synthesis, design of an Application-Specific Instruction-set Processor (ASIP), and software-driven verification approaches.

Acronyms

ADAS	Advanced Driver Assistance System
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
AXI	Advanced eXtensible Interface
CNN	Convolutional Neural Network
CPF	Common Power Format
DMA	Direct Memory Access
DSP	Digital Signal Processor
DUT	Design Under Test
ECO	Engineering Change Order
EDA	Electronic Design Automation
ESL	Electronic System Level
FFT	Fast Fourier Transform
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array

G. Martin (✉) • F. Schirrmeister • Y. Watanabe
Cadence Design Systems, San Jose, CA, USA
e-mail: gmartin@cadence.com; franks@cadence.com; watanabe@cadence.com

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_33

HLS	High-Level Synthesis
HSCD	Hardware/Software Codesign
HVL	Hardware Verification Language
HW	Hardware
IDE	Integrated Development Environment
IP	Intellectual Property
ISA	Instruction-Set Architecture
ISS	Instruction-Set Simulator
JTAG	Joint Test Action Group
LISA	Language for Instruction-Set Architectures
MAC	Multiply-Accumulator
NoC	Network-on-Chip
OFDM	Orthogonal Frequency Dependent Multiplexing
OS	Operating System
OVM	Open Verification Methodology
PCI	Peripheral Component Interconnect
PC	Personal Computer
RISC	Reduced Instruction-Set Processor
RTL	Register Transfer Level
SDK	Software Development Kit
SDS	System Development Suite
SIMD	Single Instruction, Multiple Data
SW	Software
TIE	Tensilica Instruction Extension
TLM	Transaction-Level Model
UML	Unified Modeling Language
UPF	Unified Power Format
USB	Universal Serial Bus
UVM	Universal Verification Methodology
VLIW	Very Long Instruction Word
VMM	Verification Methodology Manual
VP	Virtual Prototype
VSIA	Virtual Socket Interface Alliance
VSP	Virtual System Platform

Contents

33.1 Overview

Over the last couple of decades, the complexities of chip design have risen significantly. Where in 1995 reuse of Intellectual Property (IP) blocks was just starting and led to the foundation of the Virtual Socket Interface Alliance (VSIA) [5] in 1996, promoting IP integration and reuse, design teams are now facing the challenge of integrating hundreds of IP blocks. In 1996, most of the effort directly associated with chip design was focused on hardware itself, but since then the effort to develop software has become a budgetary item that can, depending on the application domain, dominate the cost of the actual chip development.

The Electronic Design Automation (EDA) industry responded quite early. Synopsys Behavioral Compiler, an early foray into high-level synthesis, was introduced in 1994 and Aart De Geus optimistically predicted a significant number of tape-outs before the year 2000. Gary Smith created the term Electronic System Level (ESL) in 1996, the same year that the VSIA was founded. In 1997 Cadence announced the Felix Initiative [17], which promised to make function-architecture codesign a reality. The SystemC [12] initiative was formed in 1999 to create a new level of abstraction above Register Transfer Level (RTL), but was initially plagued by remaining tied to the signal level until 2008, when the standardization of the TLM-2.0 Application Programming Interfaces (APIs) was completed. This helped interoperability for virtual platforms (also known as a Virtual Prototype (VP)) and made SystemC a proper backplane for IP integration at the transaction level. For another view on virtual prototypes, please consult ▸ Chap. 34, "Synopsys Virtual Prototyping for Software Development and Early Architecture Analysis".

When it comes to system and SoC design, at the time of this writing in 2016, the industry has certainly moved up in abstraction, but in a more fragmented way than some may have expected 20 years ago. The fundamental shortcoming of the assumptions of 1996 was the idea that there would be a single executable specification from which everything could be derived and automated. What happened instead is that almost all development aspects moved upward in abstraction, but in a fragmented way, not necessarily leading to one single description from which they can all be derived. As designers moved up in abstraction, three separate areas emerged – IP blocks, integration of IP blocks, and software.

For IP blocks, i.e., the new functions to be included into hardware and software, there is a split between IP reuse and IP development. With full-chip high-level synthesis never becoming a reality, IP reuse really saved the day, by allowing design teams to deal with complexities. It has developed into a significant market today. For IP development, there are six basic ways to implement a great new idea:

1. Manually implement in hardware.
2. Use high-level synthesis to create hardware.
3. Use an extensible or configurable processor core to create a hardware/software implementation.
4. Use tools to create a design of an Application-Specific Instruction-set Processor (ASIP).
5. Use software automation to create software from a system model.
6. Manually implement software and run it on a standard processor.

Interestingly enough, the nonmanual cases two to five all use higher-level descriptions as the entry point, but each one is different. High-level synthesis is driven by transaction-level descriptions in SystemC or C/C++, ASIPs as both IP and an associated tool flow are generated using specific language-like descriptions such as nML, Language for Instruction-Set Architectures (LISA), or the Tensilica Instruction Extension (TIE) description language [22]. Software can be auto-generated from Unified Modeling Language (UML) and MatLab/Simulink descriptions. The closest high-level unifying notations for a complete hardware/software system are SysML [10] or UML [16], as well as proprietary offerings such as MathWorks Simulink, from which both hardware blocks and software blocks can be generated automatically.

When it comes to connecting all the hardware blocks together, regardless of whether they were reused or built with one of the six options above, the user has five different options:

1. Connect blocks manually (good luck!).
2. Automatically assemble the blocks using interconnect auto-generated by ARM AMBA Designer, Sonics, Arteris, or another interconnect IP provider.
3. Synthesize protocols for interconnect from a higher-level protocol description.
4. Create a Network-on-Chip (NoC), such as a mesh NoC.
5. Use a fully programmable NoC that determines connections completely at run time.

Again, with the exception of the first (manual) and last (at run time) way to create the interconnect, the other items raise the level of abstraction. The ARM Socrates [23] and AMBA Designer environments feed information into Cadence tools such as Interconnect Workbench to set up a scenario for which performance analysis is needed, and there are specific tools to automatically create configurations of different interconnect topologies from higher-level descriptions as well.

Figure 33.1 illustrates the different methods of IP creation and integration, and the following sections of this chapter dive more deeply into two aspects – high-level synthesis using the Cadence Stratus high-level synthesis environment and the development of extensible processor cores using the Tensilica Xtensa technology. A third aspect is the software that can be found in these designs, much of it actually determining the functionality and the architecture of a chip. Efforts to achieve continuous integration of hardware and software have created what the

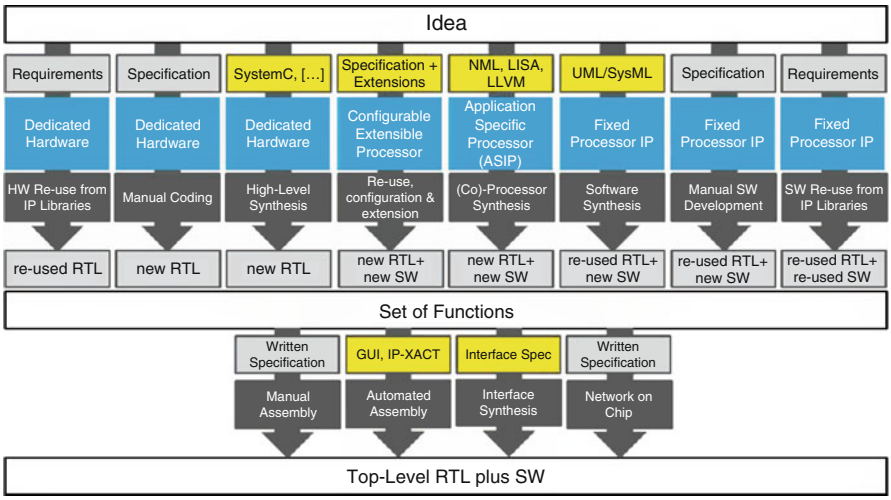


Fig. 33.1 IP creation and integration in modern chip design

industry refers to as a "shift left" – essentially, early representations or models of the hardware allowing some level of software execution to occur on the models. During a project flow today, shifting left has created various options for development vehicles on which to bring up and execute software:

1. Software Development Kits (SDKs), which do not model hardware in complete detail.
2. Virtual platforms that are register accurate and represent functionality of the hardware accurately, but without timing. Architectural virtual platforms may add cycle accuracy as a modeling style, slowing down execution, thus offering users a trade-off between speed and accuracy for architectural analysis.
3. RTL simulation is technically a representation of the hardware but is not often used for software development, unless for low-level drivers.
4. Emulation is the first platform that allows execution in the MHz range. Using emulation, users can run AnTuTu on mobile devices and bring up Linux on server chips. The intent is mainly to verify and optimize the hardware.
5. FPGA-based prototyping executes in the tens of MHz range, at times up to 100 MHz, and is a great vehicle for software development on accurate hardware.
6. The actual chip is often used in development boards to develop software.

All options except the last one use abstraction in one way or another to enable software development as early as possible. The trade-offs are time of availability during development, speed, and accuracy and the incremental effort needed for development of the development vehicle.

In many cases, hardware must take the role of executing software in the best possible way. This is why users deploy emulation and FPGA-based prototyping to

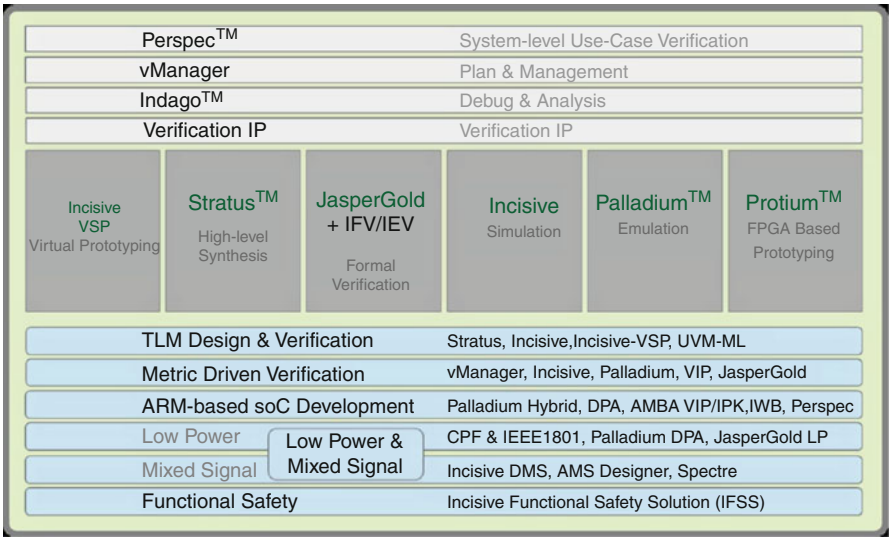


Fig. 33.2 System Development Suite

actually run mobile benchmarks like AnTuTu, as well as server benchmarks. The results help designers make changes to the design before finalizing it, to optimize performance, power, and thermal characteristics. So in a sense hardware/software codesign has become somewhat of a reality, but needs to be looked at across a family of platforms – some changes may not make it into the current design as it needs to be rolled out to meet time to market. They make it instead into the next derivative design.

Figure 33.2 shows the Cadence System Development Suite (SDS). This offers a continuous integration of development engines for verification and software development. The hardware-assisted aspects, to enable the industry demand for a shift left of software development, will be described in the following sections of this chapter.

The lessons of the last 20 years are twofold. First, no single human being is capable of comprehending all aspects of the hardware/software mix in order to generate a unified description. Complexity has simply grown too much and will continue to do so for high-end designs. The industry is just at the beginning of describing scenarios at higher levels of abstraction that can be used to allow team members with different expertise to efficiently interact. Work in Accellera on Portable Stimulus (see later details) looks promising, in defining scenarios for software-driven testing.

Second, for use cases such as performance analysis and power optimization, abstraction really has only provided a partial answer to the problem. When accuracy and predictability of the actual implementation is required, implementation really matters, to drive early design decisions, and execution at the RTL, or the level of cycle-accurate SystemC, with models abstracted from the implementation flow are predominant in order to determine power and performance. An example is a

combination of activity data gathered from RTL simulation and emulation with power characterizations abstracted from implementation representations such as .lib files, as in the combination of Cadence Palladium emulation and Cadence Joules power estimation. In contrast – for tasks such as software development of drivers in a low-power context – abstraction offers a solution using Transaction-Level Models (TLMs), sometimes combined in a hybrid fashion with RTL representations, that allows early functional verification of software, ignoring some of the detailed accuracy requirements. Examples are TLM virtual platforms annotated with low-power information and hybrid configuration of virtual platforms with emulation. As a result design teams are entering an era of both horizontal integration and vertical integration.

Horizontal integration enables verification on different engines in the flow using the same tests, sometimes referred to as "portable stimulus" as currently standardized in the Accellera working group of the same name [2]. Here are found UML-like descriptions, notations, and languages that describe scenarios. This is the next level above SystemVerilog for verification and definitely will be a hallmark of verification in the next decade, when verification shifts to the system level and designers have to rely on IP being largely bug-free. IP itself also will rise from the block level to subsystems, so the pieces to be integrated are getting bigger. The flow between the horizontal engines and hybrid engine combinations will also grow further in popularity.

Vertical integration keeps us grounded and may be the main obstacle in the way of a unified high-level design description. While in the days of the Felix Initiative, the team operated under the assumption that everything can be abstracted to enable early design decisions, it turns out that is not the case in reality. Performance analysis for chip interconnect has dropped down back to the RTL, or in the case of architectural virtual platforms, to the cycle-accurate SystemC level, simply because the pure transaction level does not offer enough accuracy to make the right performance decisions. Tools like Cadence Interconnect Workbench [13] are addressing this space today and vertically integrate higher-level traffic models with lower-level RTL and SystemC representations. The same is true for power. Abstracting power states to annotate power information to transaction-level models in virtual prototypes may give enough relative accuracy to allow development of the associated software drivers, but to get estimates accurate enough to make partitioning decisions, one really needs to connect to implementation flows and consider dynamic power. The integration of Palladium emulation with Joules power estimation from the RTL is a good example here.

Bottom line, today and for the years to come, design teams will deal with blocks to be integrated that will grow into subsystems; there will be even smarter interconnects to assemble systems on chip; and software development will have shifted left earlier. However, the separation of reuse (grown to subsystems), automatic creation (High-Level Synthesis (HLS)), and chip assembly (watch the space of integration and verification automation), plus the creation of early representations of the hardware to enable software development, will still be the predominant design techniques for very complex designs. The following sections will give more details on some of the areas touched above.

The rest of this chapter is organized as follows:

- Section 33.2 talks about the System Development Suite.
- Section 33.3 talks about virtual prototyping and hybrid execution
- Section 33.4 talks about hardware accelerated execution in emulation and FPGA-based prototyping.
- Section 33.5 talks about high-level synthesis technology.
- Section 33.6 talks about Application-Specific Instruction-set Processor technology.
- Section 33.7 talks about software-driven verification and portable stimulus.
- Section 33.8 concludes the chapter with an eye to future technology development.

33.2 System Development Suite

As indicated in the overview, a classic design flow for hardware/software projects is divided into creation, reuse, and integration of IP. Figure 33.3 shows some of the main development tasks during a project.

The horizontal axis shows the hardware-related development tasks starting with specification, IP qualification and integration, and implementation tasks prior to tape-out and chip fabrication. The vertical axis indicates development scope from hardware IP blocks through subsystems, System on Chips (SoCs), and the SoC in the actual end product (system) through software from bare-metal tasks to operating systems and drivers, middleware, and the user-facing applications.

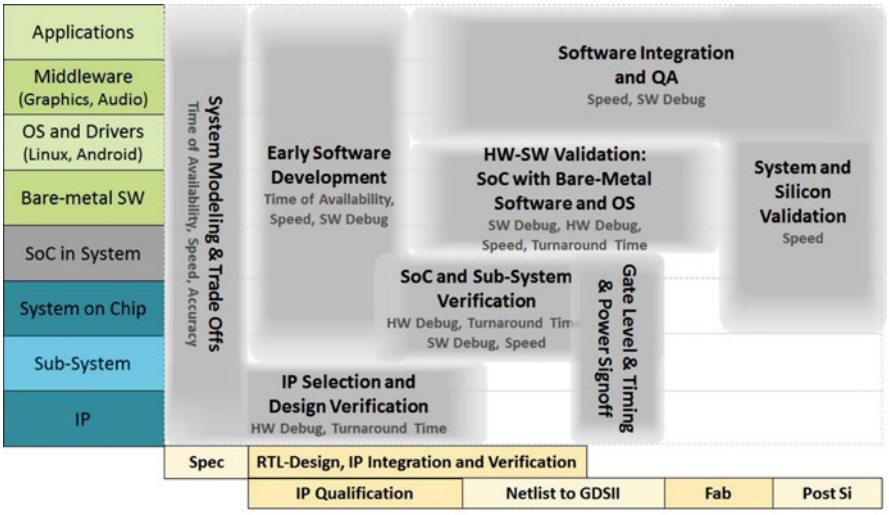


Fig. 33.3 Development tasks during a hardware/software development project

Development starts with system modeling and trade-off analysis executed by architects resulting in specifications. For system models, time of availability, speed, and accuracy are most important. Hardware development and verification is performed by hardware verification engineers for IP, subsystems, and the SoC. Initially, hardware debug and fast turnaround time are most important; once software enters the picture for subsystem verification, software debug and execution speed also become crucial. Software development happens in two main areas: hardware-aware software development for Operating System (OS) porting and utility development and application software development, requiring various levels of speed and model accuracy. The integration of hardware and software needs to be validated by HW/SW validation engineers prior to tape-out and again on silicon once actual chip samples are available. This flow can take 18–24 months; one of the major objectives is to allow agile, continuous integration of hardware and software, so developers use different execution engines and different combinations of these engines as soon as they become available.

As one can see, today's complex hardware/software designs involve many different types of developers, all with different requirements and concerns that cannot be satisfied by one engine alone. Here are the five main types of users:

1. **Application software developers** need a representation of the hardware as early as possible during a project. The representation needs to execute as fast as possible and needs to be functionally accurate. This type of software developer would like to be as independent from the hardware as possible and specifically does not need full timing detail. For example, detailed memory latency and bus delays are generally not of concern, except for specific application domains for which timing is critical.
2. **Hardware-aware software developers** would also like representations of the hardware to be available as early as possible. However, they need to see the details of the register interfaces, and they expect the prototype to look exactly like the target hardware. Depending on their task, timing information may be required. In exchange, this type of developer is likely to compromise on execution speed to gain the appropriate accuracy.
3. **System architects** care about early availability of the prototype, as they have to make decisions before all the characteristics of the hardware are defined. They need to be able to trade off hardware versus software and make decisions about resource usage. For them, the actual functionality counts less than some of the details. For example, functionality can be abstracted into representations of the traffic it creates, but for items like the interconnect fabric and the memory architecture, very accurate models are desirable. In exchange, this user is willing to compromise on speed and typically does not require complete functionality as the decisions are often made at a subsystem level.
4. **Hardware verification engineers** typically need precise timing accuracy of the hardware, at least on a clock cycle basis for the digital domain. Depending on the scope of their verification task, they need to be able to model the impact of software as it interacts with the hardware. In some cases they need to

assess mixed-signal effects at greater accuracy than standard cycle accurate RTL provides. Accuracy is considered as more important than speed, but the faster the prototype executes, the better the verification efficiency will be. This user also cares about being able to reuse test benches once they have been developed, across engines, to allow verification reuse.

5. **Hardware/software validation engineers** make sure the integration of hardware and software works as specified, and they need a balance of speed and accuracy to execute tests of significant length to pinpoint defects if they occur. This type of user especially needs to be able to connect to the environment of the chip and system to verify functionality in the system context.

Some characteristics are important to all users, but some of them are especially sensitive to some users. Cost is one of those characteristics. While all users are cost sensitive, software developers may find that a development engine may not be feasible in light of cheaper alternatives, even though the engine may have the desired accuracy or early availability in the project flow. In addition, the extra development effort that engines require beyond standard development flows needs to be considered carefully and weighed against benefits.

Figure 33.4 illustrates some of the dynamic and static development engines with their advantages and disadvantages.

The types of development engines can be categorized easily by when they become available during a project. Prior to RTL development, users can choose from the following engines:

- **SDKs** typically do not run the actual software binary but require recompilation of the software. The main target users are application software developers who do not need to look into hardware details. SDKs offer the best speed but lack accuracy. The software executing on the processors as in the examples given earlier runs natively on the host first or executes on abstraction layers like Java.

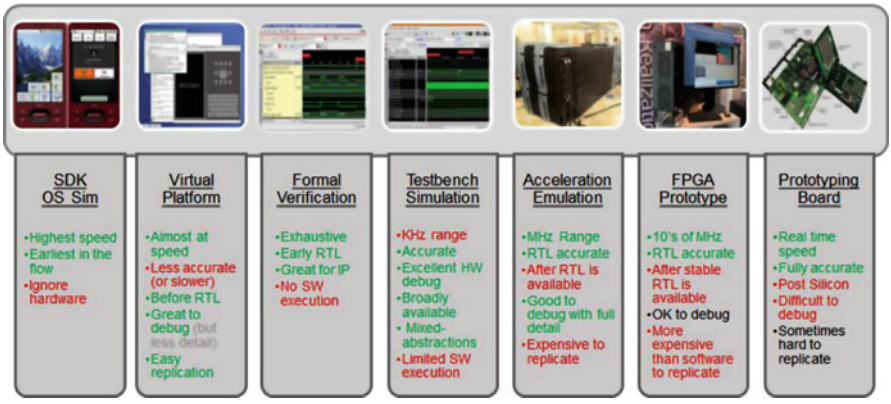


Fig. 33.4 Hardware/software development engines

Complex computation as used in graphics and video engines is abstracted using high-level APIs that map those functions to the capabilities of the development workstation.

Virtual platforms can be available prior to RTL when models are available and come in two flavors:

- **Architectural virtual platforms** are mixed accuracy models that enable architecture decision-making. The items in question – bus latency and contention, memory delays, etc. – are described in detail, maybe even as small portions of RTL. The rest of the system is abstracted as it may not exist yet. The main target users are system architects. Architectural virtual platforms are typically not functionally complete, and they abstract environment functionality into their traffic. Specifically, the interconnect fabric of the examples given earlier will be modeled in full detail, but the analysis will be done per subsystem. Execution speed may vary greatly depending on the amount of timing accuracy, but normally will be limited to tens to low hundreds of KHz. Given that cycle-accurate SystemC can be as accurate as RTL, minus sub-cycle timing annotations, automatic translation from RTL to SystemC is sometimes used: technologies like Verilator and ARM Cycle Model Studio are useful here.
- **Software virtual platforms** run the actual binary without recompilation at speeds close to real time – fifties to hundreds of MHz. Target users are software developers, both application developers and "hardware-aware software developers." Depending on the needs of the developer, some timing of the hardware may be more accurately represented. This prototype can be also used by hardware/software validation engineers who need to see both hardware and software details. Due to the nature of "just in time binary translation," the code stream of a given processor can be executed very fast, natively on the host. This makes virtual prototypes great for software development, but modeling other components of the example systems – such as 3D engines – at full accuracy would result in significant speed degradation.

Once RTL has been developed, RTL-based engines offer more accuracy:

- **RTL simulation** is the standard vehicle for hardware verification engineers. Given its execution in software, it executes slowly – in the range of hundreds of Hz – for all components in the system to be represented. It sometimes is used as an engine for lower-level software development for which great accuracy is required and appropriate length of execution can be achieved due to short simulation runs.
- **Simulation acceleration**: When RTL simulation becomes too slow, acceleration allows users to bring performance to the next orders of magnitude – 200 to 500 KHz. Acceleration is a mix of software-based and hardware-based execution. Interfaces to the real world are added, but selectively.

- **In-circuit emulation**: Now everything transitions into the emulator, test benches are synthesizable or the software executes as it will in the end product and users get even more speed – 1 to 2 MHz. Debug – especially for hardware – is great in emulation. More interfaces to the real world are added. For both *in-circuit emulation* and *acceleration*, the speed is much superior to basic RTL simulation and as such very balanced. However, when it comes to pure software execution on a processor, transaction-level models of a processor on a Personal Computer (PC) will execute faster.
- **FPGA-based prototyping**: When RTL has become mature, users can utilize Field-Programmable Gate Array (FPGA)-based platforms as even faster hardware-based execution environments. This works especially well for IP that already exists in RTL form. Real-world interfaces are now getting to even higher speeds of tens of MHz. Similarly to *acceleration* and *in-circuit emulation*, pure software execution on a processor, or transaction-level models of a processor on a PC, may still execute faster.

Finally, software development also happens on real silicon and can be split into two parts:

- Chips from the last project can be used especially for application development. This is like the SDK in the pre-RTL case. However, the latest features of the development for the new chip are not available until the appropriate drivers, OS ports, and middleware become available.
- Once the chip is back from fabrication, actual silicon prototypes can be used. Now users can run at real speed, with all connections, but debug becomes harder as execution control is not trivial. Starting, stopping, and pausing execution at specific breakpoints is not as easy as in software-based execution and prototypes in FPGA and acceleration and emulation.

To understand the benefits associated with each type of development engine, it is important to summarize the actual concerns derived from the different users and use models:

- **Time of availability during a project**: When can I get it after project start? Software virtual prototypes win here as the loosely timed transaction-level model (TLM) modeling effort can be much lower than RTL development and key IP providers often offer models as part of their IP packages. Hybrid execution with a hardware-based engine alleviates remodeling concerns for IP that does not yet exist as TLMs.
- **Speed**: How fast does the engine execute? Previous generation chips and actual samples execute at actual target speed. Software virtual prototypes without timing annotation are next in line, followed by FPGA-based prototypes and *in-circuit emulation* and *acceleration*. Software-based simulation with cycle accuracy is much slower.

- **Accuracy**: How detailed is the hardware that is represented compared to the actual implementation? Software virtual prototypes based on TLMs with their register accuracy are sufficient for a fair number of software development tasks including driver development. However, with significant timing annotation, speed slows down so much that RTL in hardware-based prototypes often is faster.
- **Capacity**: How big can the executed design be? Here the different hardware-based execution engines differ greatly. Emulation is available in standard configurations of up to several billion gates; standard products for FPGA-based prototyping are in the range of several hundreds of millions of gates, as multiple boards can be connected for higher capacity. Software-based techniques for RTL simulation and virtual prototypes are only limited by the capabilities of the executing host. Hybrid connections to software-based virtual platforms allow additional capacity extensions.
- **Prototyping development cost and bring-up time**: How much effort needs to be spent to build it on top of the traditional development flow? Here virtual prototypes are still expensive because they are not yet part of the standard flow. Emulation is well understood and bring-up is very predictable: in the order of weeks. FPGA-based prototyping from scratch is still a much bigger effort, often taking 3–6 months. Significant acceleration is possible when the software front end of emulation can be shared.
- **Replication cost**: How much does it cost to replicate the prototype? This is the actual cost of the execution vehicle, not counting the bring-up cost and time. Pricing for RTL simulation has been under competitive pressure and is well understood. TLM execution is in a similar price range; the hardware-based techniques of emulation and FPGA-based prototyping require more significant capital investment and can be measured in dollars per executed gate.
- **Software debug, hardware debug, and execution control**: How easily can software debuggers be attached for hardware/software analysis and how easily can the execution be controlled? Debugger attachment to software-based techniques is straightforward and execution control is excellent. The lack of speed in RTL simulation makes software debug feasible only for niche applications. For hardware debug the different hardware-based engines are differentiated – hardware debug in emulation is very powerful and comparable to RTL simulation, but in FPGA-based prototyping it is very limited. Hardware insight into software-based techniques are great, but the lack of accuracy in TLMs limits what can be observed. With respect to execution control, software-based execution allows one to efficiently start and stop the design, and users can selectively run only a subset of processors, enabling unique multi-core debug capabilities.
- **System connections**: How can the environment be included? In hardware, rate adapters enable speed conversion, and a large number of connections are available as standard add-ons. RTL simulation is typically too slow to connect to the actual environment. TLM-based virtual prototypes execute fast enough and virtual I/O to connect to real-world interfaces such as Universal Serial Bus (USB), Ethernet, and Peripheral Component Interconnect (PCI) have become a standard feature of commercial virtual prototyping environments.

- **Power analysis**: Can users run power analysis on the prototype? How accurate is the power analysis? With accurate switching information at the RTL level, power consumption can be analyzed fairly accurately, especially when vertically integrated with implementation flows. Emulation adds the appropriate speed to execute long enough sequences to understand the impact of software. At the TLM level, annotation of power information allows early power-aware software development, but the results are by far not as accurate as at the RTL level.
- **Environment complexity**: How complex are the connections between the different engines? The more hardware and software engines are connected (as in acceleration), the complexity can become significant and hard to handle, which needs to be weighed against the value.

Given the different types of users and their needs, the different engine capabilities, and the different concerns for the various development tasks, it is easy to see that there is no one "super"-engine that is equally suited for all aspects. Introduced in 2011, the System Development Suite is a set of connected development engines and has since then been enhanced to achieve closer integration between the engines as illustrated in Fig. 33.5.

The System Development Suite is the connection of dynamic and static verification platforms and starts with the Stratus HLS platform for IP development which is also used to raise the level of verification abstraction. The JasperGold formal verification platform is widely used throughout the flow with its different formal applications, ranging from block to SoC level. The Incisive platform for advanced verification extends from IP level to full SoCs and interacts with the Palladium acceleration and emulation platform quite seamlessly, with technologies such as hot swap between simulation and emulation.

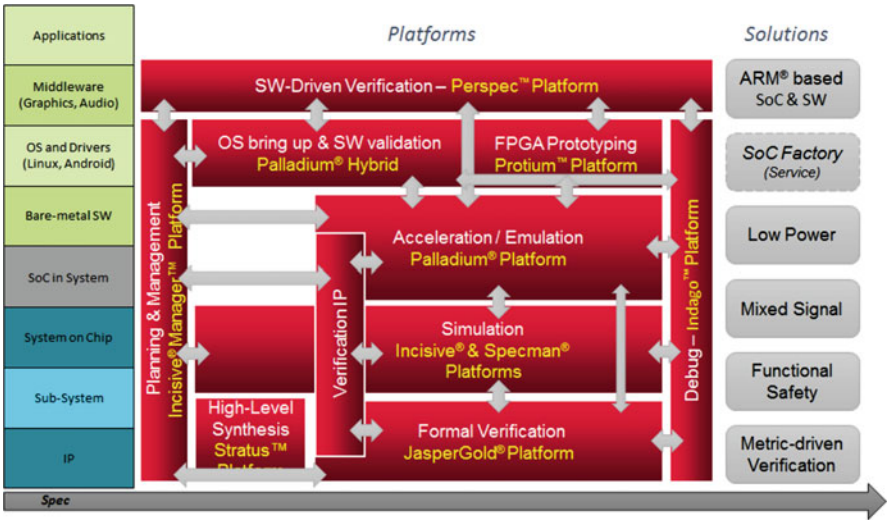


Fig. 33.5 System Development Suite engine integrations

The different engines tie together into the vManager verification center to collect and assess coverage, planning and monitoring how well verification proceeds throughout a project. Verification IP is usable across the different platforms, and with debug enabled by the Indago platform, the suite is being worked toward unified debug across the different verification engines.

Extending further into software development, the Palladium Hybrid technology connecting virtual platforms with emulation and the Protium FPGA-based prototyping technology enable software development at various levels of speed and hardware accuracy. The Perspec platform for use-case-driven verification allows the development of stimulus that is portable across the different dynamic verification engines.

Finally, there are specific solutions that combine the different engines to optimize development for ARM-based designs and low-power, mixed-signal, functional safety, and metric-driven verification. The SoC factory service enables the automation of integration and verification of IP-based designs with interfaces to IP-XACT and ARM's Socrates [23] tools.

Two system-level aspects – behavioral modeling and design space exploration – have attracted the attention of researchers for the better part of the last two decades, but so far have not become broadly supported in commercial tools. The adoption of behavioral modeling itself has been limited due to the absence of a universally accepted higher-level system language or representation. SystemC – while well adopted as an entry point for high-level synthesis and as glue for the assembly of virtual platforms, utilizing back-door interfaces as provided in SystemC TLM-2.0 APIs – has not been found suitable for higher-level descriptions. For these, proprietary techniques such as provided by National Instruments and the MathWorks and standardized entries like SysML or UML are more common. They cater to system architects and abstract both hardware and software.

In the context of the System Development Suite, SystemC is supported natively as part of multiengine simulation, while higher-level descriptions serve as references for verification with connections of MatLab/Simulink models into verification. In addition, UML style diagrams have become one option to describe system-level test scenarios to create portable stimulus that can be executed as software in multiple verification engines.

33.3 Virtual Prototyping and Hybrid Execution with RTL

Virtual prototyping was pioneered by start-ups like VasT, Virtutech, and Virtio, all of which were acquired in the last decade. It turns out that the modeling effort often is considered so high that these days "pure virtual prototypes" at the transaction level have become somewhat unusual, and mixed abstraction-level virtual prototypes, combining TLM and RTL have become predominant. Figure 33.6 shows the advantages of the different engines across the user concerns introduced in the previous section, showing clearly how the speed of virtual platforms, combined with the accuracy of RTL-based execution engines, can be advantageous.

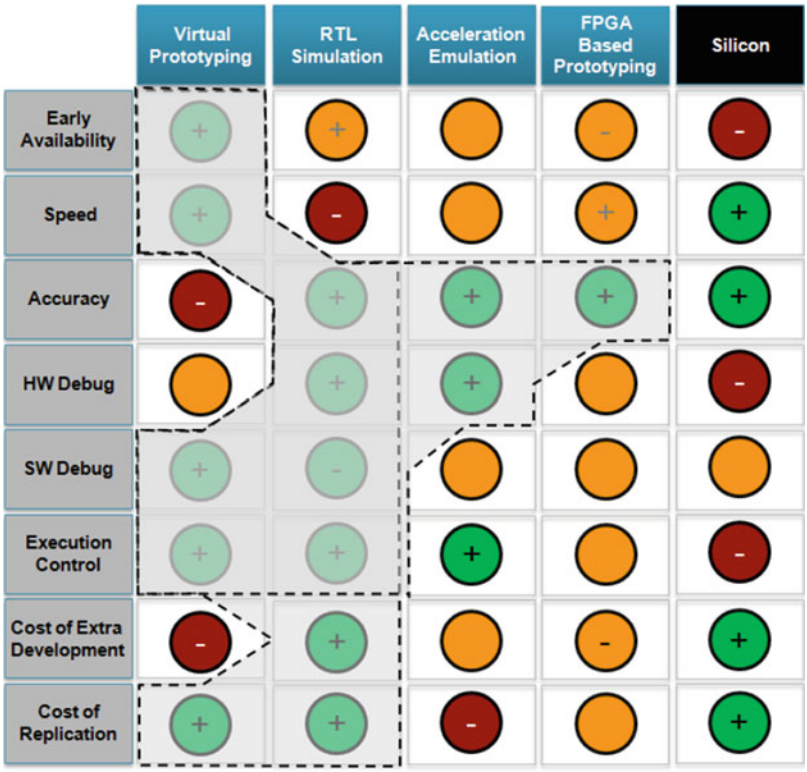


Fig. 33.6 Advantages of hybrid engine combinations

The combination of RTL simulation and virtual prototyping is especially attractive for verification engineers who care about speed and accuracy in combination. Software debug may be prohibitively slow on RTL simulation itself, but when key blocks including the processor can be moved into virtual prototype mode, the software development advantages can be utilized and the higher speed also improves verification efficiency.

The combination of emulation/acceleration and virtual prototyping is attractive for software developers and hardware/software validation engineers when processors, which would be limited to the execution speed of emulation or FPGA-based prototyping when mapped into hardware-based execution, can be executed on a virtual prototype. Equally, massive parallel hardware execution – as used in video and graphics engines – is executed faster in hardware-based execution than in a virtual prototype. For designs with memory-based communication, this combination can be very advantageous, calling graphics functions in the virtual prototype and having them execute in emulation or FPGA-based prototyping.

With hybrid techniques, users can achieve a greatly reduced time delay before arriving at the "point of interest" during execution, by using accelerated OS boot

(operating system boot-up). Billions of cycles of an operating system (OS) have to be executed before software-based diagnostics can start; therefore, OS boot itself becomes the bottleneck. The Palladium Hybrid solution combines Incisive-VSP virtual prototyping and ARM Fast Models with Palladium emulation to provide this capability.

Users such as NVIDIA [8], ARM, and CSR [20] have seen overall speedup of tests by up to ten times, when combining graphical processor unit (GPU) designs together with ARM Fast Models representing the processor subsystem. They demonstrated up to two hundred times acceleration of "OS boot," which brought them to the point of interest much faster than by using pure emulation. The actual speedup depends on the number of transactions between the TLM and RTL domains. The time to the point of interest can be accelerated significantly because during OS boot, the interaction between the TLM simulation and RTL execution in emulation (which limits the speed) is fairly limited. When the actual tests run after the OS is booted, the speedup depends again on how many interactions and synchronizations are necessary between the two domains. Some specific smart memory technology in the Palladium Hybrid solution with Virtual System Platform (VSP) and ARM Fast Models allows synchronization between both domains to be more effective (the concept can be likened to an advanced form of caching). Still, tests get accelerated the most when they execute a fair share of functionality in software.

33.4 Hardware Accelerated Execution in Emulation and FPGA-Based Prototyping

As pointed out earlier, software-based execution is limited by the number of events executed and hence has speed limitations. When considering hardware-based execution techniques, a key measure is the throughput for a queue of specific tasks, comprised of compile, allocation, execution, and debug.

Given thousands of verification and software development tasks, it is important to consider how fast the user can compile the design to create an executable of the job that then can be pushed into the execution queue. In emulation, these tasks are automated and for processor-based emulation, users compile for the latest Palladium Z1 emulation platforms at a rate of up to 140 million gates per hour, getting to results quite quickly. For simulation, the process is similar and fast. For FPGA-based prototyping, it may take much longer for manual optimization to achieve the highest speeds, often weeks if not months. Flow automation for the Protium platform, adjacent to Palladium, allows users to trade-off between bring-up and execution speed. The benefit of fast bring-up is offset by speeds between 3 and 10 MHz, not quite as fast as with manual optimization that often results in speeds of 50 MHz or more.

Allocation of tasks into the hardware platform determines how efficiently it can be used as a compute resource. For simulation farms, users are mostly limited by the number of workstations and the memory footprint. Emulation allows multiple

users, but the devil lies in the details. For large numbers of tasks of different sizes, the small granularity and larger number of parallel jobs really tips the balance here toward processor-based emulation such as the Palladium Z1 platform. In contrast the number of users per FPGA platform is typically limited to one.

The actual execution speed of the platform matters, but cannot be judged in isolation. Does the higher speed of FPGA-based prototyping make up for the slower bring-up time and the fact that only one job can be mapped into the system? It depends. As a result FPGA-based prototyping is mainly used in software development, where designs are stable and less in hardware verification. This usage is later in the cycle, but runs faster. For FPGA-based emulation, often considered faster than processor-based emulation, users have to look carefully how many jobs can be executed in parallel. And in simulation farms, the limit is really the availability of server capacity and memory footprint. The Palladium Z1 platform introduced in late 2015 is an enterprise emulation platform scalable to 9.2 billion gates for up to 2304 parallel tasks.

As the last steps of the throughput queue, debug is crucial. It is of the utmost importance to efficiently trigger and trace the debug data for analysis. FPGA-based prototyping and FPGA-based emulation slow down drastically when debug is switched on, often negating the speed advantages for debug-rich cases found when RTL is less mature. It all depends on how much debug is needed, i.e., when in the project phase the user is running the verification queue set up above. In addition, the way data is extracted from the system determines how much debug data is actually visible. Also, users need to assess carefully how the data generation slows down simulation. With processor-based emulation, debug works in a simulation-like manner. For FPGA-based systems, slowdown and accessibility of debug data need to be considered. Again, FPGA-based prototyping works great for the software development side, but for hardware debug it is much more limited compared to simulation and emulation.

As part of the System Development Suite, the Palladium platform for emulation and Protium platform for FPGA-based prototyping offer a continuum of use models as indicated in Fig. 33.7. These use models range from hardware-centric development with simulation acceleration through detailed hardware/software debug with the Palladium emulation series and faster throughput regressions as well as software-centric development with the Protium platform.

33.5 High-Level Synthesis

The history of HLS is long [18]. It was already an active research topic in the EDA community in the 1970s, and by the early 1990s it was often introduced as the "next big thing", following the significant and very successful adoption of logic synthesis. However, only recently have commercial design projects started using this technology as the primary vehicle in the hardware design flow. Even then, its commercial use was limited to design applications that were historically considered as its sweet spot, dominated by data-processing or datapath functions with little

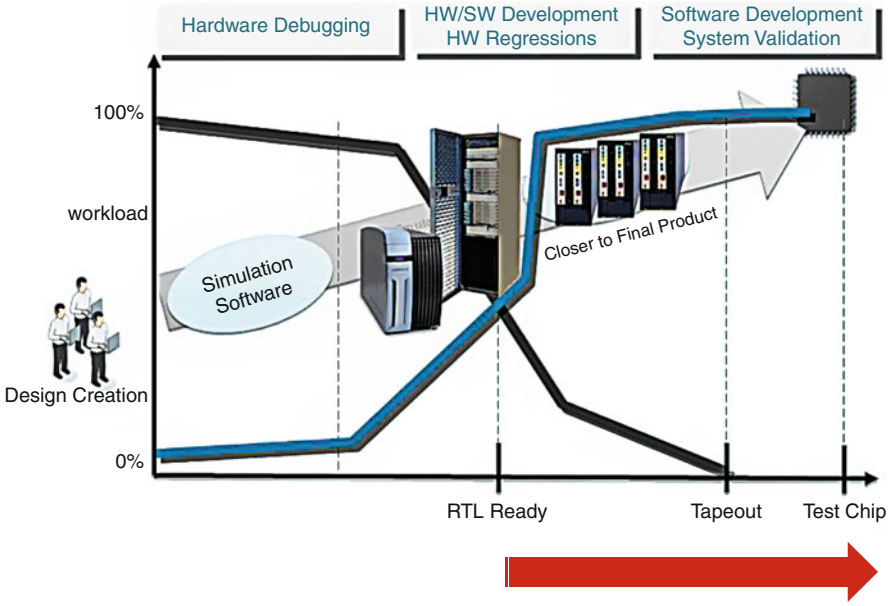


Fig. 33.7 Continuum of hardware-assisted development engines

control logic. This might suggest that HLS has had limited commercial success. On the other hand, industry users who have adopted this technology in their commercial design projects unanimously state that they would never go back to the RTL-based design flow. For them, HLS is an indispensable technology that enables them to achieve a quality of designs in tight project schedules that are not possible with RTL.

The IP blocks in today's complex designs are no longer just single datapath components, but are subsystems that include local memories for efficient data access, components that manage data transfers, and controllers for managing operations in the IPs with the rest of the system, in addition to core engines that implement algorithms to provide services defined by the IPs. These subsystems are integrated into a broad range of SoCs, which impose very different requirements in terms of implementation such as clock frequencies or performance constraints, as well as functionality on specific features or I/O interface configurations. Further, these requirements often change during the design projects. This is inevitable because nobody can foresee precisely what would be required in such complex systems before starting the projects, and details are often found when the designs are implemented or integrated into larger systems. It is therefore necessary that the design teams for those IP subsystems be able to support a broad range of design requirements imposed by different SoCs that integrate their designs, while at the same time responding to changes in requirements that arise throughout the design phases for each of them.

Stratus™ HLS, as illustrated in Fig. 33.8, addresses this need by providing three relevant characteristics that are essential for using HLS as the primary design

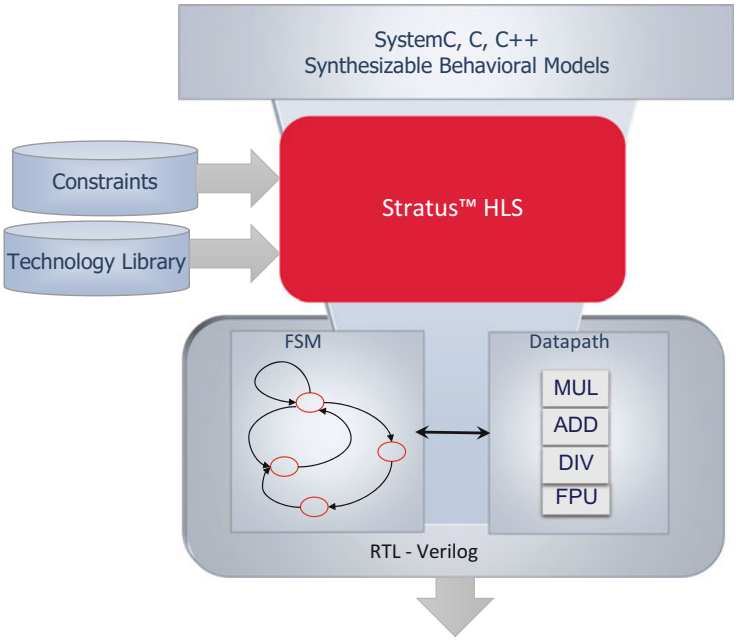


Fig. 33.8 Stratus™ HLS

technology in practice. First, it produces high-quality implementations for all components of IP subsystems that design teams need to deliver. It is no longer a tool for just datapath components. It takes as input behavioral descriptions of the target functionality in a highly configurable manner. The descriptions are specified using the SystemC language, where standard C++ techniques are used to define features and microarchitectures that can be included in the design through simple reconfiguration of the same descriptions. It also takes design requirements of the target implementations and technology library and produces synthesizable RTL for downstream implementation processes.

The breadth of configurations one can achieve with these descriptions is far beyond what is possible with RTL or parameterized RTL models, because the behavioral descriptions for Stratus HLS can result in totally different RTL structures just by changing the design parameters. The level of abstraction of these behavioral descriptions allows the designers to specify their design intent by focusing only on a few key specifics of the architectures while leaving the tool to figure out all the other details automatically. With this, they can easily evaluate various architectural choices of not only individual components of the IP but the whole subsystem.

For example, in achieving high-performance hardware implementations of algorithms, it is often important to take into account not only the cost of implementing the arithmetic computation of the algorithms but also the impact of accessing the data required for the algorithms. To address this concern, designers evaluate

the architecture of the memory hierarchy. In RTL design, they typically consider allocation of data to the memory hierarchy in such a way that data required by the individual arithmetic operations can be located close to the resources for executing the operations. This kind of exploration is easy in HLS, where one can change the memory hierarchy using design parameter configurations and data allocation to specific type of memories can be decided automatically.

The second aspect with which Stratus HLS provides strong value to IP design teams is the integration of this technology with the rest of the design and verification flow. Since HLS produces implementation from abstracted behavioral descriptions, it inevitably lacks detailed information that becomes available only in subsequent phases of the implementation flow. This causes a risk in general that design decisions made by HLS could cause issues that are difficult to close later in the design process. To mitigate this risk, one could either incorporate downstream tools within HLS or establish a closed loop from those tools back to an HLS tool. Stratus HLS does both. It uses the logic synthesis engine during the optimization process, so that it makes design decisions by accurately taking into account the information of actual resources implemented by logic synthesis. To cope with the wire congestion issue, the tool provides a back annotation mechanism to correlate the resources that cause high congestion during the layout phase to objects in the input behavioral descriptions, so that the designer can evaluate the root causes of wire congestion quickly.

The HLS design flow is also required to work with existing RTL designs, so that if the components designed with HLS are adjacent to components already written in RTL, the connections between them must be done seamlessly, despite the fact that they are written in different languages and using different abstraction levels for the interfaces. Stratus HLS provides features that automatically produce interlanguage interface adapters between the behavioral and RTL components. The user can decide on simulation configurations of a design that have mixtures of HLS components and RTL components, and the tool automatically inserts the adapters to establish the necessary connections. Such a mixture of behavioral and RTL descriptions also arises within a component that is fully designed with HLS.

Typically, a behavioral description for the component is written in a hierarchical manner, so that the design can be implemented gradually. When designers analyze the quality of implementation, they often focus on a particular subcomponent, leaving the rest of the design either at the behavioral level or at RTL depending upon the progress of the design phase. Stratus HLS provides a capability where the user can define multiple architectural choices in the individual subcomponents and then specify for each of them whether they want to use the behavioral description in simulating the subcomponent or the RTL description made for a particular architectural choice defined for it. The tool then automatically synthesizes the subcomponents as specified and combines the resulting RTL with behavioral descriptions of the remaining subcomponents to produce a simulation image. With this, the user can seamlessly verify the functionality of the component while focusing on particular subcomponents to explore various architectural choices to produce a high-quality implementation.

The third aspect that is extremely important for the adoption of HLS in practice is the support for Engineering Change Orders (ECOs). In the context of HLS, the main concern is support for functional ECOs, at a late stage, when design components have already been implemented to the logic or layout level and verification has been done, and the need arises to introduce small changes in the design functionality. In the RTL-based design flow, the designers carefully examine the RTL code and find a way to introduce the changes with minimal and localized modification of the code. If the designer tries to do the same with HLS, by introducing small changes in the behavioral description, when HLS is applied to the new description, the generated RTL often becomes very different from the original one. The logic implemented in RTL may change very significantly even if the functionality is very similar to the original one.

Stratus HLS provides an incremental synthesis feature to address this issue. In this flow, the tool saves information about the synthesis of the original design, and when an ECO happens, it takes as input this information together with the newly revised behavioral description. It then uses design similarity as the main cost metric during synthesis and produces RTL code with minimal differences from the original RTL code while meeting the specified functionality change.

High-quality implementations obtained from highly configurable behavioral descriptions for the whole IP subsystem, the integration with the existing design and verification flow, and the support for ECOs are the primary concerns that one needs to address when adopting high-level synthesis technology for designing new components of IPs. The fact that major semiconductor companies have successfully adopted Stratus HLS as an indispensable technology in their critical design projects is attributed to its strong capabilities in these aspects.

More information on HLS capabilities can be found in [6].

33.6 Application-Specific Instruction-Set Processors

This section discusses the concept of an ASIP and relates them specifically to hardware/software codesign. This concept is used to develop a particular codesign methodology: "processor-centric design." For another view on ASIPs, see ▸ Chap. 12, "Application-Specific Processors".

33.6.1 ASIP Concept and Tensilica Xtensa Technology

The foundation for processor-centric subsystem design is configurable, extensible processor technology, which has been developed by a number of academic and commercial groups since the 1990s [14, 22]. Tensilica technology [15, 27] dates from the late 1990s and has been applied to a wide variety of ASIP designs.

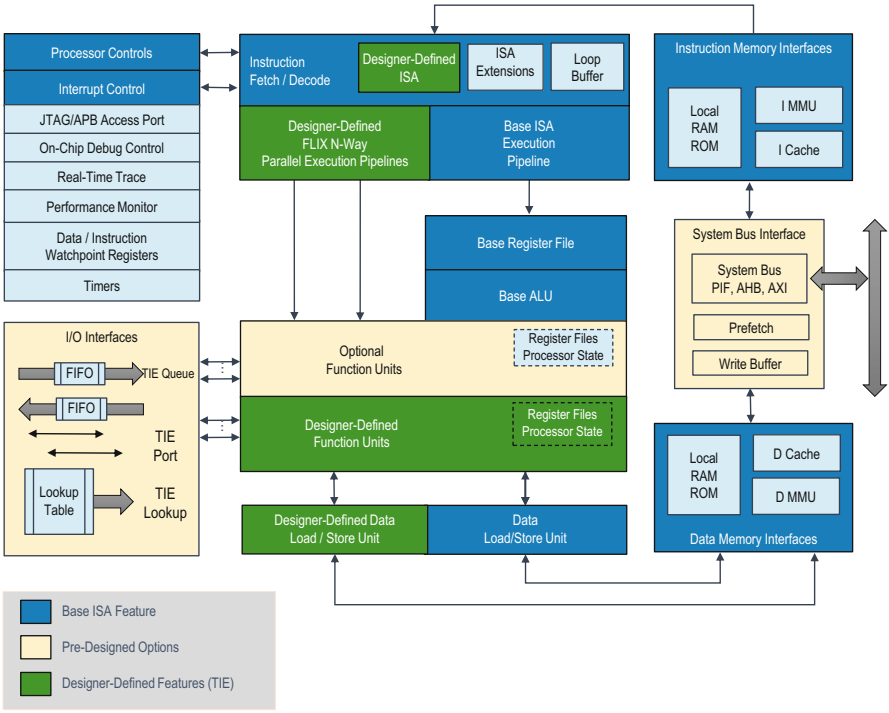


Fig. 33.9 Xtensa ASIP concept

Configurable, extensible processors allow designers to configure structural parameters and resources in a base Reduced Instruction-Set Processor (RISC) architecture as shown in Fig. 33.9. Extensibility allows design teams to add specialized instructions for applications. Automated tool flows create the hardware and software tools required, using specifications for structural configuration, and instruction extensions, defined by an architectural description language [21].

Configurable structural architecture parameters include:

- Size of register files
- Endianness
- Adding functional units, e.g., Multiply-Accumulators (MACs) and floating point
- Local data and instruction memory interfaces including configurable load-store units and Direct Memory Access (DMA) access and memory subsystem configuration
- Instruction and data cache attributes
- System memory and bus interfaces including standard buses such as Advanced eXtensible Interface (AXI)

- Debug, tracing, Joint Test Action Group (JTAG)
- Timers, interrupts and exceptions
- Multi-operation Very Long Instruction Word (VLIW) operation bundling
- Pipeline depth and microarchitecture choice
- Port, queue, and lookup interfaces into the processor's datapath

Instruction extensions, defined in Tensilica's TIE language [30], define specialized register bank width and depth, special processor state, operations of almost arbitrary complexity, their specification and optimized hardware implementation, SIMD-width, encoding, scheduling (single or multi-cycle), usage of operands and register ports, and bundling into multi-operation VLIW instructions. In addition, a number of documentation descriptions and software properties that influence operation scheduling in the compiler can be defined in TIE. Other aspects of the user programming model using instruction extensions, such as support for new C-types and operator overloading, and mapping of instruction sequences into a single atomic operation or group of operations can also be defined in TIE. Aggressive use of parallelism and other techniques in user-defined TIE extensions can often deliver 10X, 100X, or even greater performance increases compared to conventional fixed instruction-set processors or Digital Signal Processors (DSPs).

The automated tool flow generates the tooling for compilers, assemblers, instruction-set simulators, debuggers, profilers, and other software tools, along with scripts for optimized hardware implementation flows targeting current Application-Specific Integrated Circuit (ASIC) technologies [3].

Xtensa technology has been developed for over 17 years, from the founding of Tensilica as a separate company and its acquisition in 2013 [27]. This technology has been extensively verified [4, 24]. Designers perform their optimization and create their ideal Xtensa processor by using the Xtensa processor generator. In addition to producing the processor hardware RTL [11, 31], the Xtensa processor generator automatically generates a complete, optimized software-development environment. Two additional deliverables with Xtensa are:

1. Xtensa Xplorer Integrated Development Environment (IDE), based on Eclipse, which serves as a cockpit for single- and multiple-processor SoC hardware and software design. Xtensa Xplorer integrates software development, processor optimization, and multiple-processor SoC architecture tools into one common design environment. It also integrates SoC simulation and analysis tools.
2. A multiple processor (MP)-capable Instruction-Set Simulator (ISS) and C/C++ callable simulation libraries, along with a SystemC development environment XTSC.

ASIPs support Hardware/Software Codesign (HSCD) methodologies, albeit not quite in the classical sense of "all Hardware (HW)" vs. "all Software (SW)". ASIPs allow the computation and communications required by particular algorithms and applications to be mapped into flexible combinations of classical SW and application-oriented operations which are tuned to the application requirements.

Algorithms which are control-dominated can be mapped into an ASIP which is like a classical RISC machine, with configurability limited to aspects such as the memory subsystem and debug attributes. Algorithms heavy on computation with many application-specific operations can be mapped into ASIPs with extensive instruction extensions that greatly reduce the number of cycles required to execute and as a corollary, reduce the overall energy consumption of the algorithm by a large fraction. Algorithms heavy on communications methods or needing ancillary hardware execution units can utilize the port, queue and lookup interfaces to both simplify and improve the performance possible in passing data and control information from one core to another or to adjunct hardware blocks.

In this sense, ASIPs explode the design space exploration possibilities available to designers. They no longer need to live with just hardware or just selecting one from a list of predefined processor cores. They can tune one processor or a group of homogeneous or heterogeneous processors specifically to the particular application domain and algorithms their design is focused on. A good overview of design space exploration using Xtensa processors can be found in Chap. 6 of [4]. Design space exploration is discussed using this concept of processor-centric design.

33.6.2 DSP Design Using Xtensa

Xtensa ASIP technology has been applied by customers to create their own application-specific processors. It has also been applied internally within the research and development teams to create DSPs tuned to particular application domains. The key domains addressed through the years have been audio processing, communications, and video, imaging, and vision processing applications.

Audio [19] has been for many years a major focus of ASIP technology and audio DSPs. Several variations of audio DSPs exist, with distinct tradeoffs of power, speed performance, area, and cycle-time performance. As a result, the family of audio DSPs allow distinct hardware/software tradeoffs to be made by choosing the optimal audio DSP for a particular requirement. Software audio codecs and audio post-processing applications are also an important part of the offering.

A video codec subsystem called 388VDO [7,9] was developed several years ago. This consisted of two DSPs: a stream processor and a pixel processor, with adjunct DMA block, and an optional front-end Xtensa control processor. Several video encoders and decoders were offered as software IP with this subsystem, supporting major standards (such as MPEG2, MPEG4, JPEG, H264) and resolutions up to D2. The design of the Instruction-Set Architecture (ISA) for the two DSPs was done in close collaboration with the software team developing the video codecs and drew heavily on the concepts of hardware/software codesign, profiling, and performance analysis.

More recently, advanced vision and image processing processors [26, 29], are applicable to a wide variety of applications, have been developed. Computer vision is one of the fastest-growing application areas as of 2016, with particular attention being paid to Advanced Driver Assistance System (ADAS) in automotive

and security applications, gesture recognition, face detection, and many more. For another view on embedded computer vision and its relationship to ASIPs, see ▸ Chap. 40, "Embedded Computer Vision".

In the communications domain, a focus on wireless baseband processing was the impetus for development of specialized configurable DSPs [25]. In fact a family of DSPs was developed using a common ISA, variable Single Instruction, Multiple Data (SIMD) widths (16, 32, and 64 MACs) and a scalable programming model, based on evolving an earlier 16 MAC baseband DSP [28]. The basis is the combination of a real and complex vector processor, with specialized instructions for FFT for Orthogonal Frequency Dependent Multiplexing (OFDM).

33.6.3 Processor-Centric Design and Hardware/Software Design Space Exploration

This section describes the processor-centric design approach enabled by configurable, extensible ASIP methodologies, drawing on details to be found in Chap. 6 of [4].

Processor-centric design is a family of design approaches that includes several alternative methodologies. What is common to all of them is a bias toward implementing product functionality as software running on embedded processor(s), as opposed to dedicated hardware blocks. This does not mean that there are no dedicated hardware blocks in a processor-centric design; rather, these blocks are present as a matter of necessity rather than choice. In other words, dedicated hardware blocks will be present in the design where they *must* be, rather than where they *could* be. This could be to achieve the required level of performance, to achieve the desired product cost target, or to minimize energy consumption.

Traditional fixed ISA processors offer very stark tradeoffs for embedded product designers. They are generic for a class of processing and have few configurability options to allow them to be tailored more closely to the end application. The rise of ASIPs meant that designers could no longer consider the use of fixed embedded processors for an increasing number of the end-product application. ASIPs can now offer enough performance and sufficiently low energy consumption, at a reasonable cost, to take over much of the processing load that would have heretofore relied on dedicated hardware blocks. Thus ASIPs have been a key development enabling a much more processor-centric design style.

Traditional fixed ISA processors can be simply divided into control- and data-plane processors. Control processors, such as ARM and MIPS cores, are often used for non-data intensive applications or parts of an application, such as user interfaces, general task processing, high-level user applications, protocol stack processing, and the like. Data-plane processors are often fixed ISA DSPs that have special instructions and computational and communications resources that make them more suitable for data-intensive computation, especially for real-time signal and image processing.

As demonstrated earlier, ASIPs have grown in variety, number, and importance in recent years. Because an ASIP can be configured and extended to optimize its performance for a specific application, ASIPs offer much greater performance (say, 10–100X) and much lower energy consumption (perhaps half to one-quarter) than the same algorithm compiled for a fixed-ISA standard embedded processor – even a DSP. There are a few simple reasons to account for this advantage:

1. ASIPs allow coarse-grained configuration of their basic structure to better match the particular applications. If an application is mainly control processing, an ASIP may offer a fairly basic instruction set, but if an application is mainly intensive data processing (e.g., from the "data plane") – for example, audio, video, or other image processing – it may offer special additional instructions (zero-overhead loops, MACs) tuned to media or DSP kinds of applications.
2. The size and widths of registers can be tuned to be appropriate for the particular application domain.
3. Interfaces, such as memory interfaces, and caches can be configured or left out of the design dependent on data and instruction locality and the nature of the underlying algorithmic data access patterns. Sometimes caches may be more effective than local instruction and data (scratchpad) memories; sometimes the opposite may be the case.
4. Memory or bus interfaces may also be configured as to width and protocol – e.g., AMBA AHB or AXI.
5. Diagnosis and debug features such as trace ports, JTAG interfaces, and the like may be added or left out.
6. Interrupts and exception handling may be configured according to design need. Often the elaborate exception recovery mechanisms used in general purpose processors may be unnecessary in an ASIP tuned to run a very specific algorithm deeply embedded in a system.
7. VLIW style multi-operation instructions may be added to processors to support applications with a large amount of irregular instruction-level parallelism that can take advantage of such features.
8. SIMD type instructions – e.g., 2-, 4-, 8-, 16-way, or larger – may be added to processors to support vector-style simultaneous instructions acting on large chunks of data at a time.
9. Instructions may be tuned to specific algorithmic requirements. For example, if two 13-bit quantities need to be multiplied in an inner loop that dominates an algorithm, use of a 32-bit multiplier is both wasteful of area and energy and possibly performance.
10. Fine-grained instruction extensions including instruction fusions drawn from very specific algorithmic code can lead to significant increases in performance and savings in power. For example, a sequence of arithmetic operations in a tight loop nest that might account for 90% of the cycles in executing the algorithm on a data sample may be replaced with a single fused instruction that carries out the sequence in one or a few clock cycles.

Use of ASIPs instead of general purpose processors can lead, for known algorithms, to a radical improvement in performance and power consumption. This is true whether an ASIP is totally designed to support one very specific algorithm or if it is designed to support a class of applications drawn from a single domain. A specialized audio processing ASIP could be designed just to support MP3 decoding or could be slightly generalized so that it will support many different audio codecs – possibly optimizing one codec such as MP3 that is very widely used, but with general audio instructions added so that new codecs can still take advantage of the specific instructions and hardware incorporated in the ASIP.

Sometimes the complexity of a specific application domain may lead to a heterogeneous multi-processor, multi-ASIP design as being optimal for a certain target range of process technologies. Video codecs, baseband, vision, and imaging are examples.

A processor-centric design methodology needs to support design space exploration when deciding whether particular functional requirements for a design can be mapped to a single fixed ISA processor running at a suitable rate, a multi-processor implementation (such as a cache-coherent symmetric multi-processing "multi-core" cluster), a special fixed ISA processor such as a DSP, a single ASIP, a set of ASIPs configured to work together as a heterogeneous multi-processor subsystem, a combination of fixed ISA processor(s) and ASIP(s), and finally, mapping any part of the function into dedicated hardware blocks, almost certainly working in conjunction with the processors. A wide range of communications architectures, from shared memory accessed via buses through dedicated local memories, DMA blocks to permit concurrent data and instruction movement, direct communications such as First-In First-Out (FIFO) queues between processors and from processors to hardware, and NoCs may be used. In general, the processor-centric design flow has the following steps:

1. Start with an algorithm description. This is often reference C/C++ code obtained from a standards organization. Alternatively, it may be a reference code generated from an algorithmic description captured in a modeling notation such as the MathWorks' MatLab or Simulink, or in UML or one of its profiles, and using code generation to obtain executable C or C++.
2. Characterize the algorithm by running it on a very generic target processor. This will give designers some idea of the general computational and communications requirements of the algorithm (communications being defined as both data access and control access communicating into and out of the algorithm).
3. Identify "hot spots" in the target application. These will very often be loop nests in which multiple instructions are executed over large data samples. Techniques such as instruction fusion (combining multiple instructions into one); vectorization (SIMD) methods, where the same instruction is applied to many data items; and multi-operation instructions – where several operations without dependencies could be executed simultaneously on a VLIW-style architecture – are commonly identified.

4. Configure the processor and add instruction extensions to accelerate the execution of the algorithm. Re-characterize the code running on the new modified target. It may be necessary to restructure the code or insert pragmas into it in order that the compiler can take full advantage of vectorization (SIMD) or fused instructions.
5. If the performance targets for the algorithm are met and the estimates of power consumption and cost (area in terms of gates) are satisfactory, stop: this processor is now a reasonable choice for the function. Otherwise, further code restructuring and further configuration exploration and additional instruction extensions may be important. In this case, repeat the last few steps until either a satisfactory result is achieved, or it is necessary to add specialized hardware blocks as coprocessors in order to achieve the desired results.
6. If hardware blocks are necessary, they may be created using high-level synthesis tools, based on the algorithmic description for that part of the algorithm which must migrate to hardware. The design team may explore a variety of mechanisms for tying such accelerating blocks to the main processor – hardware FIFOs, coprocessor interfaces, or loosely coupled with systems buses, or DMA.

33.7 Software-Driven Verification and Portable Stimulus

The industry is rapidly approaching a new era in dynamic verification as indicated in Fig. 33.10.

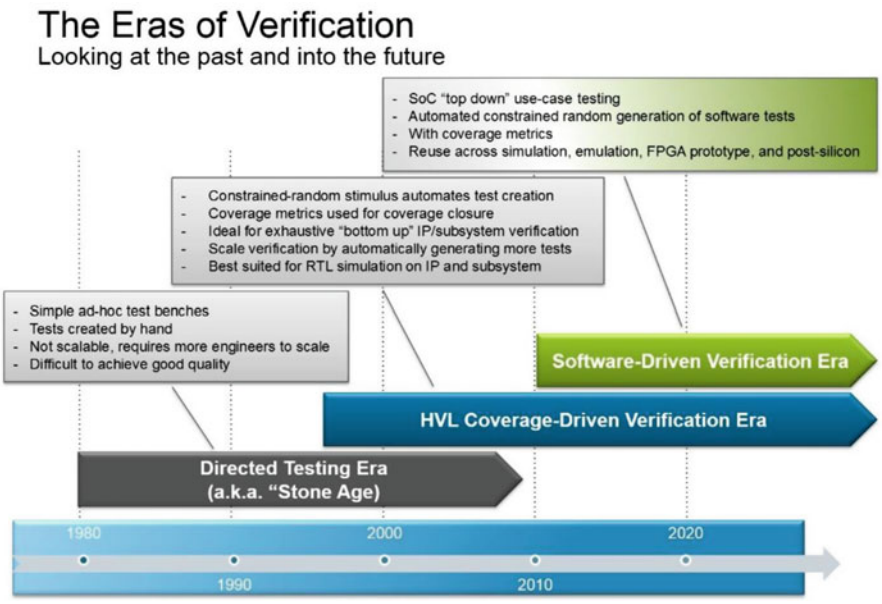


Fig. 33.10 The eras of verification

In the early days of verification, the "Stone Age" directed testing dominated verification. Design and verification engineers, at the time still emerging, were developing simple ad hoc test benches and creating tests by hand. This approach was not very scalable, as it required more engineers when more verification was required. As a result, it was very difficult to achieve good quality, and the confidence in how to get there and whether everything was verified was very hard to achieve.

In synchronization with the era of heavy IP reuse – sometime in the late 1990s to the early 2000s – the era of Hardware Verification Languages (HVLs) began. This is where specific verification languages such as VERA, ***e***, Superlog, and eventually SystemVerilog fundamentally changed the verification landscape. Methodologies were developed, including the Verification Methodology Manual (VMM), Open Verification Methodology (OVM), and later Universal Verification Methodology (UVM). In this era of verification, constrained-random stimulus automated test creation and coverage metrics were introduced to measure coverage closure. The level of automation involved in this era allowed users to scale verification by automatically generating more tests and made the HVL-based approaches ideal for exhaustive "bottom-up" IP and subsystem verification.

By 2016 the objects to be verified – modern SoCs – have evolved. They now contain many IP functions, from standard I/Os to system infrastructure and differentiating IP. They include many processor cores, both symmetric and asymmetric, both homogeneous and heterogeneous. Software executes on these processors, from core functionality such as communication stacks and infrastructure components such as Linux and Android operating systems all the way to user applications. Experts seem to agree that the UVM, while great for verification of IP blocks, falls short for SoC verification. The two main reasons are software and verification reuse between execution engines. It is important to note that UVM will not likely go away – it is fine for the "bottom-up" IP and some subsystem verification – and will continue to be used for these applications. However, UVM does not extend to new approaches for "top-down" SoC-level verification.

When switching from bottom-up verification to top-down verification, the context changes. In bottom-up verification, the question to verify is how the block or subsystem behaves in its SoC environment. In top-down verification, the correctness of the integrated IP blocks itself is assumed, and verification changes to scenarios describing how the SoC behaves in its system environment. An example scenario may be "view a video while uploading it." On top of the sequence of how the hardware blocks in the system interact, this scenario clearly involves a lot of software.

This is where traditional HVL-based techniques run up against their limits. They do not extend well to the software that is key to defining scenarios. Scenarios need to be represented in a way that they can be understood by a variety of users, from SoC architects, hardware developers, and software developers to verification engineers, software test engineers, and post-silicon validation engineers. They need to be comprehended by a variety of different users to allow efficient sharing. Also, the resulting test/verification stimulus needs to be portable across different

verification engines and even the actual silicon once available, enabling horizontal reuse. Software executing on the processors in the system – called software-driven verification – is the most likely candidate. Third and finally, the next wave of verification needs to allow both IP integration as well as IP operation within its system context to be tested, i.e., vertical reuse.

The Perspec System Verifier platform is one means to achieve portable stimulus by means of software-driven verification. Consider a use case from above: "view a video while uploading it." This six-word statement translates into bare-metal actions at the SoC level that need to be executed in a form such as "take a video buffer and convert it to MPEG4 format with medium resolution using any available graphics processor. Then transmit the result through the modem via any available communications processor and, in parallel, decode it using any available graphics processor and display the video stream on any of the SoC displays supporting the resulting resolution."

The state space this scenario creates is vast. Various resolutions, different video algorithms, different resources, different types of memory buffers, etc. need to be considered. Writing such a test manually, if even feasible, is hard to do and requires valuable system knowledge. And then the resolution, memory, or resources change – which makes it harder. This is where UML-like use-case definitions and constrained-random solving techniques to instantiate data and control flow with valid combinations of parameters come in as shown in Fig. 33.11.

The abstract use case reads from a memory buffer, converts data into a second memory buffer, and then in parallel transmits and decodes for display. The UML-based description is intuitive and can be understood by the various stakeholders. The automation involved transforms this description into an actual UML activity diagram with randomized video buffers, specific choices of video conversion

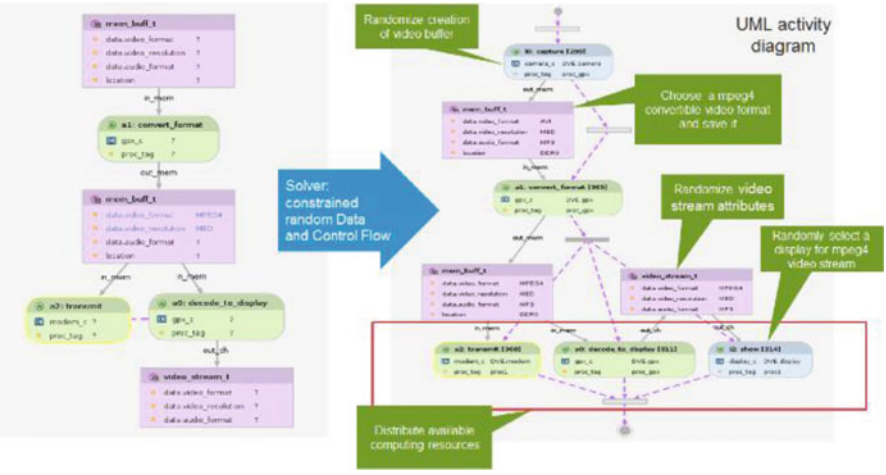


Fig. 33.11 UML use-case definition in Perspec System Verifier

formats such as MPEG4 to save into specific buffers, randomized video stream attributes, random selection of a display for the stream playback, and distribution across available compute resources.

Perspec System Verifier automatically generates the associated tests that execute on the processors in the design and run them on the various validation engines of the System Development Suite – from virtual platforms through RTL simulation, emulation, FPGA-based prototyping, and the actual silicon.

33.8 Conclusion

This chapter has surveyed a number of technologies for hardware/software codesign and coverification. They are undergoing constant evolution, and new applications are being found for these various technologies as technology and design practices evolve and change. This snapshot from 2016 represents state of the art in these areas as of that time. There are several new technology directions being explored.

Xtensa technology will evolve in two ways. First, it will support a wider range of microarchitectural choices and features, giving users even more options for creating ASIPs that meet their application needs. Secondly, it will be used in new and emerging application domains to offer new types of DSPs to users. Vision processing is a hot area in 2016 and likely to remain so, especially for emerging automotive applications. An even hotter subset of vision processing is the use of "AI" or deep-learning techniques such as Combinational Neural Networks and variations to support automotive ADAS applications. Both general vision DSPs with special Convolutional Neural Network (CNN) capabilities and highly application-specific CNN or other neural network ASIPs are possibilities.

System Development Suite (SDS) continues to move toward closer integration of different verification engines. The concept of "Continuum of Verification Engines" (COVE) [1] has been publicly discussed. Further connection of virtual prototyping, high-level synthesis, formal verification, RTL simulation, emulation, and FPGA-based prototyping have been recent and ongoing trends:

- **Verification acceleration**: Connection of the verification computing platform and RTL simulation to achieve accelerated execution is second only to *in-circuit emulation* applications. The Design Under Test (DUT) resides on the emulator and the test bench on the host; the host execution of the test bench controls the overall speed and users report 200–300X speedup over pure simulation.
- **Simulation/emulation hot swap**: This is a unique capability with SDS; users can run in one environment for a certain time, stop, and switch to the other.
- **Virtual platform/emulation hybrid**: This allows teams to reduce the time to the point of interest using, for example, fast models from ARM in virtual platforms connected to emulation.
- **Multi-fabric compilation for hardware engines**: In SDS, users have a multi-fabric compiler that can target both emulation and FPGA-based prototyping for in-circuit emulation, avoiding lengthy reengineering.

- **Unified Power Format (UPF)/Common Power Format (CPF) Low-Power Verification**: Power verification using either standard can be run in emulation and RTL simulation, for example, to verify the switching on and off of various power domains.
- **Portable stimulus**: Enables reuse of verification across various engines, including the chip itself. Vertical reuse from IP to subsystems to full SoCs is possible. Finally reuse is possible across various engineering disciplines.
- **Interconnect performance analysis**: Integrates verification IP (VIP) and RTL simulation to enable performance optimization and verification for interconnect.

A key capability going forward will be the automation of integration in a general way – for example, Interconnect Workbench (IWB) automatically generates test benches targeting different platforms.

References

1. http://www.deepchip.com/items/0549-04.html
2. Andrews M, Hristov B (2015) Portable stimulus models for c/SystemC, UVM and emulation. In: Design and verification conference and exhibition (DVCON). Accellera Systems Initiative
3. Augustine S, Gauthier M, Leibson S, Macliesh P, Martin G, Maydan D, Nedeljkovic N, Wilson B (2009) Generation and use of an ASIP software tool chain. In: Ecker W, Müller W, Dömer R (eds) Hardware-dependent software: principles and practice. Springer, Berlin, pp 173–202
4. Bailey B, Martin G (2010) ESL models and their application: electronic system level design and verification in practice. Springer, Boston
5. Bailey B, Martin G, Anderson T (eds) (2005) Taxonomies for the development and verification of digital systems. Springer, New York. The Virtual Socket Interface Alliance lasted from 1996 to 2008 but its archival web site is no longer functional as of 2015. This book may be all that is left of its work
6. Balarin F, Kondratyev A, Watanabe Y (2016) High level synthesis. In: Scheffer L, Lavagno L, Markov I, Martin G (eds) Electronic design automation for integrated circuits handbook, vol 1, 2nd edn. CRC Press/Taylor and Francis, Boca Raton
7. Bellas N, Katsavounidis I, Koziri M, Zacharis D (2009) Mapping the AVS video decoder on a heterogeneous dual-core SIMD processor. In: Design automation conference user track. IEEE/ACM. http://www.dac.com/46th/proceedings/slides/07U_2.pdf
8. Bianchi M, Snyder T, Grabowski D (2015) Denver IP acceleration leveraging enhanced debug. In: CDNLive silicon valley. Cadence Design Systems
9. Ezer G, Moolenaar D (2006) Mpsoc flow for multiformat video decoder based on configurable and extensible processors. In: GSPx Conference
10. Friedenthal S, Moore A, Steiner R (2014) A practical guide to SysML, 3rd edn. Morgan-Kaufmann, Boston
11. Gonzales R (2000) Xtensa: a configurable and extensible processor. IEEE Micro 20(2):60–70
12. Grötker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic Publishers, Dordrecht
13. Heaton N, Behar A (2014) Functional and performance verification of SoC interconnects. Embed Comput Des. http://embedded-computing.com/articles/functional-performance-verification-soc-interconnects/
14. Ienne P, Leupers R (2006) Customizable embedded processors: design technologies and applications. Morgan Kaufmann/Elsevier, San Francisco
15. Leibson S (2006) Designing SOCs with configured cores: unleashing the Tensilica Xtensa and diamond cores. Morgan Kaufmann/Elsevier, San Francisco

16. Martin G, Müller W (eds) (2005) UML for SoC design. Springer, Heidelberg
17. Martin G, Salefski B (2001) System level design for SoC's: a progress report – two years on. In: Ashenden P, Mermet J, Seepold R (eds) System-on-chip methodologies and design languages. Springer, Heidelberg, pp 297–306
18. Martin G, Smith G (2009) High-level synthesis: past, present, and future. IEEE Des Test 26(4):18–25
19. Maydan D (2011) Evolving voice and audio requirements for smartphones. In: Linley mobile conference. The Linley Group
20. Melling L, Kaye R (2015) Reducing time to point of interest with accelerated os boot. In: CDNLive silicon valley. Cadence Design Systems
21. Mishra P, Dutt N (2006) Processor modeling and design tools. In: Scheffer L, Lavagno L, Martin G (eds) Electronic design automation for integrated circuits handbook, vol 1, 1st edn. CRC Press/Taylor and Francis, Boca Raton
22. Mishra P, Dutt N (eds) (2008) Processor description languages. Elsevier-Morgan Kaufmann, Amsterdam/Boston
23. Murray D, Boylan S (2013) Lessons from the field: IP/SoC integration techniques that work. In: Design and verification conference and exhibition (DVCON). Accellera Systems Initiative
24. Puig-Medina M, Ezer G, Konas P (2000) Verification of configurable processor cores. In: Proceedings of design automation conference (DAC). IEEE/ACM, pp 184–188
25. Rowen C (2012) Power/performance breakthrough for LTE advanced handsets. In: Linley mobile conference. The Linley Group
26. Rowen C (2015) Instruction set innovation in fourth generation vision DSPs. In: Linley processor conference. The Linley Group
27. Rowen C, Leibson S (2004) Engineering the complex SoC: fast, flexible design with configurable processors. Prentice-Hall PTR, Upper Saddle River
28. Rowen C, Nuth P, Fiske S, Binning M, Khouri S (2012) A DSP architecture optimised for wireless baseband. In: International symposium on system-on-chip (ISSOC)
29. Sanghavi H (2015) Baby you can drive my car: vision-based SoC architectures. In: Linley processor conference. The Linley Group
30. Sanghavi H, Andrews N (2008) TIE: an ADL for designing application-specific instruction set extensions. In: Mishra P, Dutt N (eds) Processor description languages. Elsevier-Morgan Kaufmann, San Francisco
31. Wang A, Killian E, Maydan D, Rowen C (2001) Hardware/software instruction set configurability for system-on-chip processors. In: Proceedings of design automation conference (DAC). IEEE/ACM, pp 184–188

Synopsys Virtual Prototyping for Software Development and Early Architecture Analysis

34

Tim Kogel

Abstract

This chapter summarizes more than 20 years of experience by the virtual prototyping group of Synopsys in the commercial deployment of Hardware/Software Codesign (HSCD). The goal of HSCD has always been to reduce time to market, increase design productivity, and improve the quality of results. From all the different facets of HSCD, virtual prototyping – complemented by links to emulation and FPGA prototyping – has so far proven to achieve the best return of investment with respect to these goals. This chapter first gives an overview of the main virtual prototyping use cases in the context of an end-to-end prototyping flow, which also includes physical prototyping and hybrid prototyping. The second part introduces the SystemC Transaction-Level Model (TLM) standard and the Unified Power Format (UPF) as the main modeling languages for the creation of Virtual Prototypes (VPs) and system-level power models. The main body of this chapter focuses on the commercially deployed virtual prototyping use cases for architecture exploration and system-level power analysis.

Acronyms

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AT	Approximately Timed
AT-BP	Approximately Timed Base Protocol
AV	Architects View
AXI	Advanced eXtensible Interface
CA	Cycle Accurate
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access

T. Kogel (✉)
Synopsys, Inc., Aachen, Germany
e-mail: tim.kogel@synopsys.com

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_34

DMI	Direct Memory Interface
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
ECU	Electronic Control Unit
FPGA	Field-Programmable Gate Array
FT	Fast Timed
GFRBM	Generic File Reader Bus Master
GPU	Graphics Processing Unit
HAPS	High-performance ASIC Prototyping System
HDL	Hardware Description Language
HSCD	Hardware/Software Codesign
HW	Hardware
IP	Intellectual Property
ISS	Instruction-Set Simulator
LT	Loosely Timed
MCO	Multi-Core Optimization
MPSoC	Multi-Processor System-on-Chip
OS	Operating System
PMU	Power Management Unit
QoS	Quality of Service
RFTS	Run Fast Then Stop
RTL	Register Transfer Level
SCML	SystemC Modeling Library
SLP	System-Level Power
SMP	Symmetric Multi-Processing
SoC	System-on-Chip
SW	Software
TCL	Tool Command Language
TLM	Transaction-Level Model
UPF	Unified Power Format
VPU	Virtual Processing Unit
VP	Virtual Prototype

Contents

34.1 Introduction

In a traditional development process, the hardware/software integration and validation can only start after the hardware development is finished and the first silicon samples are available. This forces a sequential dependency between the Hardware (HW) and Software (SW) development phases and puts a lot of stress on the overall schedule. Prototyping offers a set of methodologies to overcome this dependency by "shifting left" the architecture design and software development flows.

Today, the following variants of prototypes are in deployment by semiconductor and electronic system companies:

- **Virtual prototypes** are fast, executable models of the System-on-Chip (SoC). They are typically created from SystemC Transaction-Level Models (TLMs) that are delivered by semiconductor Intellectual Property (IP) companies and are extended with models that are specifically created for the SoC in development (see Sect. 34.2.1). Different types of Virtual Prototypes (VPs) are created based on the requirements for simulation speed and timing accuracy: Synopsys offers Platform Architect for Multi-Core Optimization (MCO) for the creation and usage of VPs for architecture design as well as Virtualizer™ for software development related use cases.
- **Physical prototypes** provide specialized FPGA-based systems and tools to execute Register Transfer Level (RTL) implementations at high speeds. This way, Field-Programmable Gate Array (FPGA) prototypes are useful for system validation and software development purposes. Synopsys provides the HAPS® high-performance Application-Specific Integrated Circuit (ASIC) prototyping system.
- **Hybrid prototypes** combine a VP with a physical prototype; see also Sect. 3 in the ▸ Chap. 37, "Control/Architecture Codesign for Cyber-Physical Systems". For target use cases like IP driver development, hybrid prototypes offer users a way to optimize the prototyping setup based on the availability of TLMs and RTL implementations. Taken together, Synopsys Virtualizer™ and HAPS® provide a hybrid prototyping environment.

All three types of prototypes typically require a dedicated team inside a semiconductor organization that specializes in providing these prototypes to the actual end users.

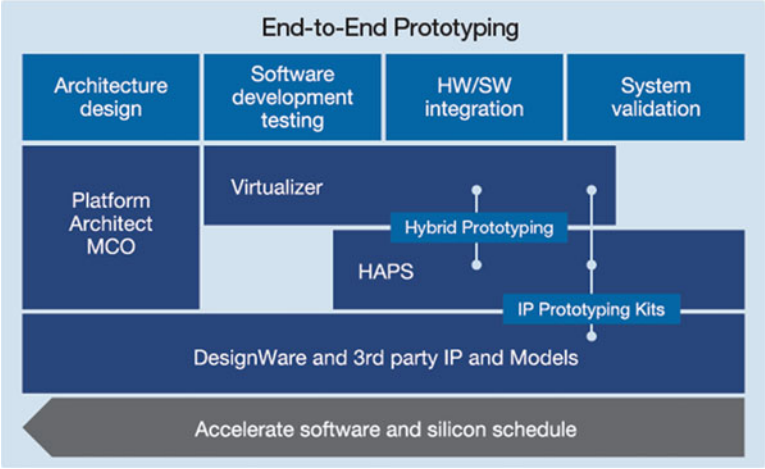


Fig. 34.1 Design tasks and solutions in end-to-end prototyping

As depicted in Fig. 34.1, these prototyping methods can be applied for multiple tasks in a software-driven SoC design flow: architecture design, software development and testing, HW/SW integration, and system validation. The biggest value is achieved when they are applied across all the stages of SoC design.

The following paragraphs review these tasks individually.

34.1.1 Architecture Design

Architect teams are typically working in several stages and have projects that deal with generation N+2, where N refers to the current SoC generation in production. Traditionally, architects rely on past experience and static spreadsheet analysis for estimating power and performance of the next- or second-next-generation product. This manual and static analysis is becoming increasingly difficult due to the increasing complexity of electronic products. Virtual prototyping enables architects to simulate the impact of critical application use cases and of specific design decisions on the overall performance and power consumption; see also the paragraph on architectural virtual platforms in Sect. 2 of ▶ Chap. 33, "Hardware/Software Codesign Across Many Cadence Technologies".

To benefit from early architecture exploration using a VP, an architect needs three fundamental ingredients: (1) performance models, (2) power models, and (3) application scenarios.

Performance models describe relevant components of the SoC, such as the interconnect and memory subsystems, with sufficient accuracy to enable critical design decisions based on the latency and throughput data provided by simulating these performance models.

Power models are needed to also analyze the expected power and energy consumption for the main components that consume power on the SoC. These power

models rely on power modeling standards [11] augmented with power consumption data from IP data sheets and with measurements during the implementation steps from previous projects.

Finally, the processing and communication requirements of the software that will actually run on the chip needs to be described at an abstract level in terms of an application workload model. Virtual prototyping offers ways to either manually capture these task-based workload models or to extract them from software execution traces.

The result is that the performance and power of the architecture can be explored more accurately compared to relying on static spreadsheet-based exploration. Thanks to their flexibility and high simulation speed, VPs enable the simulation of orders of magnitude more architecture variations compared to doing the same at the RTL. This typically leads to double-digit gains in terms of power and performance trade-offs, reducing the cost and excessive power consumption of over-designed products. Section 34.3 elaborates more on the different aspects of virtual prototyping for architecture analysis and optimization.

34.1.2 Software Development and Testing

The software development schedule can achieve the biggest time to market gains by applying prototyping. Starting software development earlier and, thus, shortening the overall schedule as well as enabling earlier feedback between hardware and software teams is a real game changer.

For many years, semiconductor companies have been using only physical prototyping to do software development, typically for IP- and subsystem-related software. Most of them have adopted commercial FPGA-based prototyping solutions to benefit from existing design and debug automation tools to be able to achieve the fastest time to first prototype and optimize for highest performance (see also Sect. 4 in ▸ Chap. 33, "Hardware/Software Codesign Across Many Cadence Technologies"). The market is now shifting rapidly to using integrated commercial solutions, comprised of hardware and software tools to achieve a high-performance prototype in weeks rather than months, hence, significantly increasing the useful time of prototyping before silicon arrival.

With the advent of larger FPGAs, these physical prototyping systems are usable to prototype much larger portions of the SoC, including Graphics Processing Units (GPUs), enabling more software development early in the design cycle. While FPGA-based prototyping has proven to provide high value by enabling early software development, virtual prototyping has been adopted by many semiconductor vendors as a complimentary prototyping solution. By creating and deploying VPs, semiconductor vendors are able to further shift left their software development and start more than 12 months before silicon availability.

Where FPGA-based prototyping offers key benefits enabling HW/SW integration and especially system validation, virtual prototyping provides key capabilities to accelerate software development and scale software testing. Since VPs are based on models, they are not dependent on RTL implementation availability, and, hence,

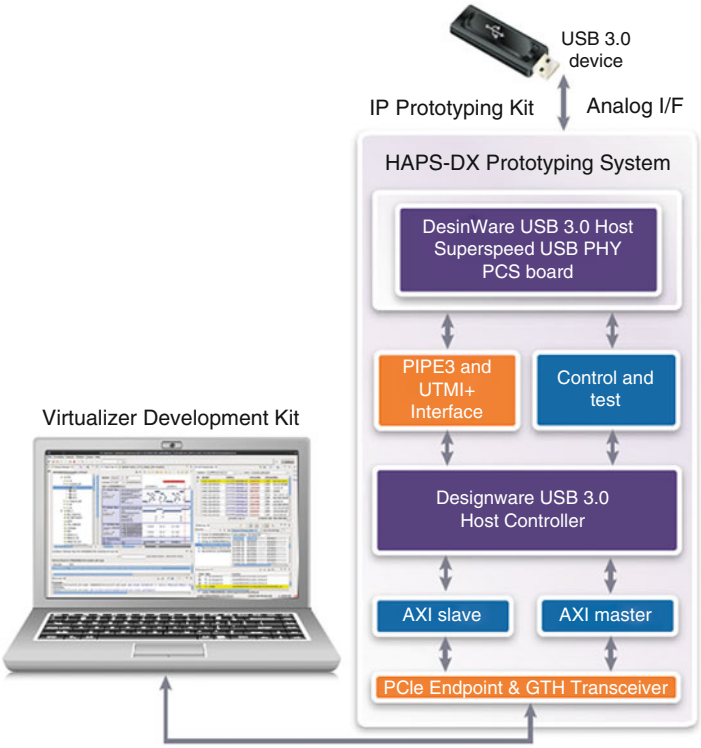


Fig. 34.2 Example DesignWare hybrid IP prototyping kit

they enable software development much earlier in the design cycle. Another benefit of using models rather than RTL execution on a physical prototype is the improved control and debug visibility provided by VPs. They also allow for fault injection to test how the system will react [21], and they scale more easily to massive parallel testing and, thus, help increase software quality.

As shown in Fig. 34.2, virtual and physical prototyping environments can be merged into a hybrid prototyping environment, which offers a great solution for IP-specific software bring up. Early availability of models and thus VPs for new processors combined with IP mapped on an FPGA offer an early and functionally complete solution for IP-specific software development, testing, and HW/SW integration.

34.1.3 Hardware/Software Integration and System Validation

Hardware/software integration typically starts at the subsystem level on an FPGA-based prototype, when RTL IP blocks or subsystems that are considered to be relatively stable are integrated with firmware or drivers. The bring-up time can

be significantly reduced when VPs or hybrid prototypes are used for the software development so that the software is more or less functionally complete by the time of the HW/SW integration step.

System validation deals with removing the uncertainty of how the SoC with the software stack will perform in the target environment. The physical prototype is plugged into the real-world environment and performs realistic scenarios. Examples of this can be system validation of a networking SoC inside an actual networking device for which execution speeds of multiple 10s of MHz are required. Or validation of application processor SoCs that have to support many different interface protocols such as USB, eMMC, and others.

34.1.4 System-Level Power Analysis

Energy proportional computing is a key concern for many electronic products, most notably for any battery-driven mobile consumer device [6]. Any device should consume only as much power as absolutely required to perform a certain task. The increasing number of Central Processing Units (CPUs) with their high frequencies, the increasing size of LCD screens and cameras, and the multitude of radios and sensors are driving the total power consumption beyond what is acceptable by the consumer. Only proper management of the power states within these devices allows minimizing the total power consumption.

Both architecture and software have a significant impact on power consumption:

- The key decisions impacting power consumption are taken during the architecture definition phase. For example, a relevant use case for a smartphone could be the streaming of a video via the cellular network and displaying it in HD resolution on a connected LCD screen. Each of the use cases requires the services of a certain set of components. This use case analysis drives the partitioning of the device into power domains and their respective operating points. Typically, the supply voltage and frequency of the power domains can be controlled individually to provide the flexibility to later optimize the power dissipation and energy consumption of the different use cases. The optimal definition of power domains and operating points is key to achieve the goal of an energy proportional system.
- Software plays a significant role in the device power management at run time. The software controls and drives hardware components which actually consume power. Software stacks such as Android/Linux with millions of lines of code implement various power saving strategies on almost each software layer starting at the driver and ending up in the application layer [7]. A software power inefficiency or malfunction can quickly cause a 5× drop in standby time.

Power consumption is an orthogonal aspect, which caters to all prototyping use cases. Therefore the modeling of the power consumption should be as much as

possible independent of the actual prototyping itself. Section 34.2.3 shows how to add component-level power models as an overlay to a VP based on the IEEE 1801-2015 UPF-3.0 standard [11]. Using such power models enables architects as well as software developers to take the impact of their design decisions and software implementation on the power consumption into account.

34.1.5 Summary

By leveraging the different technologies in the context of an end-to-end prototyping solution, the typical design time line is significantly reduced.

The shift left impact of end-to-end prototyping not only reduces the overall design time line and hence the time-to-market but also helps companies that are deploying this methodology validate that their products better match the original design requirements. By optimizing the architecture early on in context of the software scenarios and designing the hardware and software side by side, the resulting product is better balanced.

By now, virtual prototyping for early software development is already a widely deployed methodology [4, 23]. Refer also to Sect. 2 in ▸ Chap. 33, "Hardware/Software Codesign Across Many Cadence Technologies". Therefore the main body of this chapter focuses more on the virtual prototyping use cases for architecture exploration and system-level power analysis. Before going there, the next section first introduces the modeling methodologies, which are the foundation for creating VPs.

34.2 Modeling for Virtual Prototyping

Modeling is the key initial task for creating a VP. This task requires the specification of the system or SoC to be modeled, the modeling tools, and knowledge of modeling languages. This section first focuses on SystemC TLMs, which are the established lingua franca for the creation of VPs. The second part gives an introduction to UPF-3.0, the new modeling standard for system-level power analysis.

34.2.1 The SystemC Transaction-Level Modeling Standard

The IEEE 1666 standard for SystemC and TLM defines the widely accepted modeling language for the creation of VPs [12]. SystemC is a C++ library providing a set of classes to model system components and their communication interfaces, plus a cooperative multitasking environment to model concurrent activity in a system. On top of SystemC, the TLM library supports a modeling style where the communication interfaces between system components is not based on individual signals, but on a set of function calls and a payload representing the full semantics of the communication interface. This reduces the number of synchronization points

between communicating component models, which in turn greatly improves the overall speed of the event-driven SystemC simulation kernel. Since 2008, the TLM-2.0 standard provides a well-defined set of Application Programming Interfaces (APIs) and payload constructs to create interoperable TLMs for memory-map-based communication protocols.

The IEEE Std 1666 TLM-2.0 Language Reference Manual [12] identifies the following coding styles:

- The **Loosely Timed (LT)** modeling style aims to maximize the simulation speed of a model by abstracting the communication to the highest level and by minimizing the synchronization overhead.
- The **Approximately Timed (AT)** modeling style focuses on the timing of the transactions between different components in a system by providing multiple timing points for each transaction.

As illustrated in Fig. 34.3, both LT and AT modeling styles use the same concept of sockets, generic payload, and an extension mechanism for modeling memory-mapped communication protocols. The extension mechanism allows adding of custom attributes to the generic payload, which is important to model protocol-specific attributes, like the transaction id in the Advanced eXtensible Interface (AXI) protocol [1]. This common infrastructure enables the smooth integration of models using different modeling styles. On the other hand, LT and AT leverage specific mechanisms to cater to the specific requirements of different virtual prototyping use cases for software development and architecture analysis.

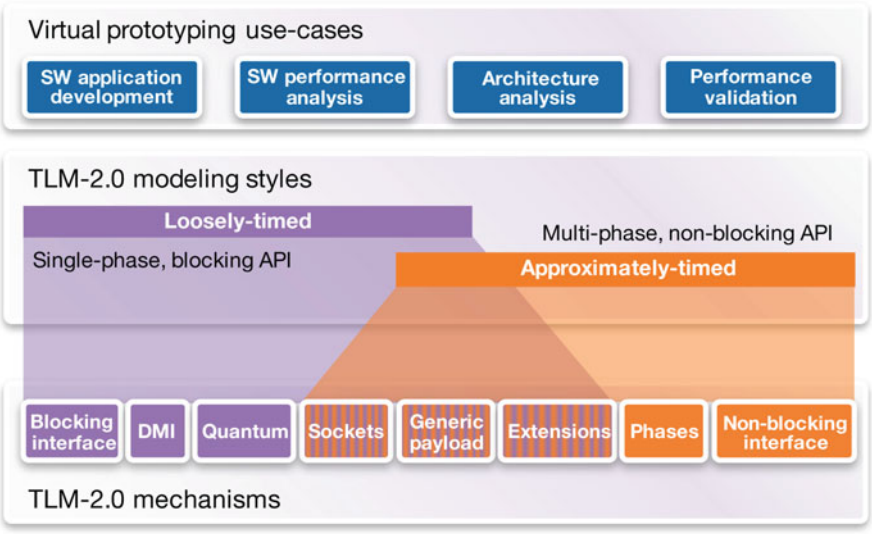


Fig. 34.3 TLM-2.0 modeling styles and mechanisms

34.2.1.1 Loosely Timed Modeling Style

VPs for software development are created in order to provide an abstract model of the target hardware platform, which can execute the unmodified software. The key requirements are:

- Simulation speed: It is important that the VP can execute software at a speed that is as close as possible to real time of the actual target device.
- Register accuracy: In order to run embedded software correctly, the memory and memory-mapped register layout and content should be modeled.
- Functional fidelity: All relevant responses of the target hardware should be modeled.

The LT modeling style is intended to maximize the execution speed while providing the minimal level of timing fidelity. The key concepts in the TLM-2.0 standard to achieve high simulation speed on top of the event-driven SystemC simulation kernel are temporal decoupling, the Direct Memory Interface (DMI), and blocking communication:

- Temporal decoupling allows initiator components, like processor models, to run ahead of the global time for a maximum quantum of time before synchronizing with the SystemC kernel; see also Sect. 1.3 in ▸ Chap. 19, "Host-Compiled Simulation".
- DMI allows initiator components to bypass the regular TLM interface and directly access instruction and data memory via the simulation host address.
- For non-DMI access to memories and peripheral registers, the simple blocking TLM transport interface is used. As depicted on the left side of Fig. 34.4, LT communication is modeled using a single function call.

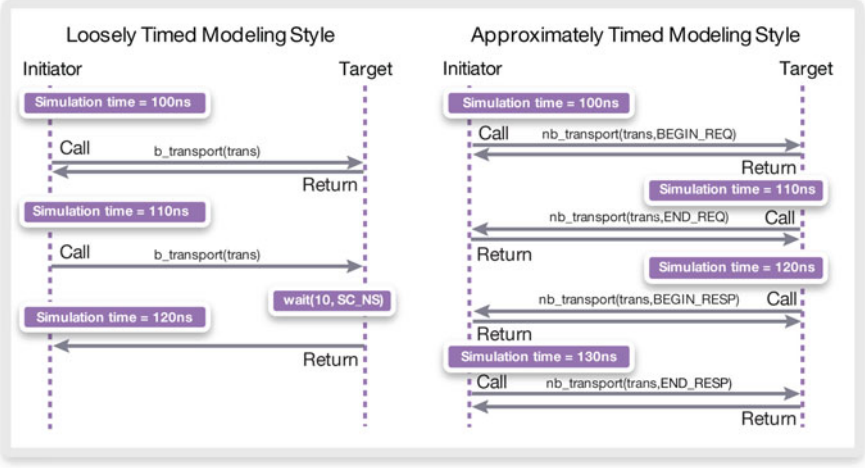


Fig. 34.4 TLM-2.0 Loosely Timed (*left*) and Approximately Timed (*right*) protocols

The LT modeling style reflects the hardware interactions with software and how register content is updated. For example, timer interrupts happen roughly at the intended time to simulate the timing calibration loop in a Linux boot or to execute real-time software in automotive Electronic Control Units (ECUs).

34.2.1.2 Extended Loosely Timed Modeling Style

The TLM-2.0 generic payload only covers the common subset of transaction attributes like address, data, and burst length. The extension mechanism allows to include additional protocol-specific attributes to the generic payload, e.g., security extensions, atomic transactions, coherency flags, etc. Based on this extension mechanism, owners of on-chip bus protocols have defined a layer on top of TLM-2.0 for creating protocol-specific models in a interoperable way [2].

The loosely timed modeling style has been very successful in fostering the availability of interoperable models from all major IP providers [3, 25]. The availability of LT TLMs for off-the-shelf IP blocks has significantly reduced the investment for creating VPs for software development.

34.2.1.3 Approximately Timed Modeling Style

VPs for early architecture analysis and exploration are created in order to provide an abstract model of the target hardware, which reflects relevant performance metrics, e.g., bandwidth, throughput, utilization, and contention. The key requirements are:

- Scalable timing accuracy: The accuracy requirements depend on the goal of the project. For example, an abstract model of a DRAM is good enough for exploring HW/SW partitioning, but a highly accurate model is needed for optimizing the configuration of the DRAM memory controller.
- Compositional timing: The end-to-end performance of a system can be obtained from assembling a set of components which only model their individual timing.

Compared to the LT modeling style described previously, the AT modeling style is intended to model the communication with more detailed timing. As shown on the right side of Fig. 34.4, a single transaction is broken into multiple phases to reflect the timing of a bus protocol in more detail. The non-blocking TLM transport interface is used to mark start and end of each phase.

The TLM-2.0 initiator and target sockets bundle a forward and backward path in one interface to enable bi-directional communication. The initiators calls nb_transport to mark the begin of a request phase and sometime later the target calls nb_transport to mark the end of a request phase. As depicted in Fig. 34.4, the TLM-2.0 standard defines an Approximately Timed Base Protocol (AT-BP) with a request and response phase marked by four distinct timing points. This enables the modeling of basic communication aspects like throughput, latency, and transaction pipelining.

34.2.1.4 Extended Approximately Timed

The TLM-2.0 AT-BP has limited expressiveness when it comes to accurately representing any real-life on-chip bus protocols:

- It does not provide with timing points for the individual data beats of a burst transfer. This becomes particularly problematic when interfacing TLM-2.0 AT-BP with Cycle Accurate (CA) or RTL models.
- It requires all address and data information to be available for writes at the start of the transaction.
- It is not possible to have concurrent read and write requests, as required by, e.g., the AMBA AXI protocol [1].

To overcome these deficiencies of the AT-BP, the TLM-2.0 standard provides an extension mechanism for the AT modeling style, which enables the definition of additional protocol phases and timing points. Together with the extension mechanism for the generic payload, which is also used for loosely timed modeling, this enables the more accurate modeling of on-chip bus protocols. In fact, the AT extension mechanism allows the definition of fully CA modeling of real-life bus protocols.

The issue is that protocol-specific extensions break the interoperability between AT-BP and extended AT models. Synopsys has defined a Fast Timed (FT) modeling infrastructure. FT is based on the TLM-2.0 AT mechanism and enables the definition of more accurate protocols while preserving interoperability with the AT-BP:

- Each protocol extends the generic payload with an attribute indicating the current state in the protocol state machine.
- For each protocol, protocol-specific attributes are added as needed, e.g., for cacheability, out-of-order transactions, etc. This should be limited to those attributes that are not already covered by the TLM-2.0 AT-BP. These extensions are ignorable in the sense that a model should assume they have a default value in case they are not present in the payload.

The idea is that FT protocols remain compatible with the TLM-2.0 AT-BP and rely on extended sockets and payload to provide the necessary protocol conversion logic so that conversions are only done when required and can be inserted automatically.

34.2.1.5 Summary

The goal of the IEEE 1666 TLM-2.0 standard is to enable model interoperability at the level of SoC building blocks, e.g., processors, buses, memories, peripherals. For this purpose, TLM-2.0 standardizes the modeling interface for memory-mapped bus communication, which is the prevalent SoC interconnect mechanism. The LT and AT modeling styles cater to the different requirements of different use cases like software development and architecture analysis. Although AT allows more detailed timing modeling than LT, the modeling style of the communication interface should

not be confused with the abstraction level or timing accuracy of a model itself. LT and AT only refer to the communication aspect, whereas abstraction and timing accuracy also depend on the timing and granularity of the structure and behavior inside the component.

34.2.2 Modeling Objects and Patterns

Thanks to the TLM-2.0 interoperability standard, today many TLM-2.0 compliant models of standard SoC components like processors, buses, and memories are available from the respective IP provider. However, there are still a significant number of custom building blocks, e.g., timers, interrupt controllers, Direct Memory Access (DMA) controllers, or HW accelerators, for which specific models need to be created. TLM-2.0 defines the interoperability standard, but it does not prescribe how to model the internal behavior. In order to reduce the actual modeling effort, a well-defined modeling methodology and a library of reusable modeling objects is required. In larger companies, this is especially important to unify the modeling style across distributed modeling teams.

This section explains the concept of modeling objects and patterns based on the publicly available SystemC Modeling Library (SCML) from Synopsys [27].

34.2.2.1 The SystemC Modeling Library (SCML)

SCML is a layer on top of SystemC and TLM-2.0. It hides a lot of the complexity and common code that is required to correctly manage TLM-2.0 transactions, and it provides with modeling objects that handle common aspects of VP modeling. The modeling objects in the SCML promote the separating communication, behavior, and timing [14]. This way, the models created based on this methodology support different modeling styles like LT, AT, and FT.

Figure 34.5 illustrates the coding style, which is enabled by the SCML modeling objects:

- The interface to the interconnect model is separated from the actual behavior of the component. For the behavior of the component, a generic TLM-2.0 compliant LT, AT, or FT bus interface can be used.
- The interface between the extended protocol and the generic TLM-2.0 protocol used by the SCML storage objects is implemented by a protocol adaptation layer.
- The actual behavior of the component can be separated into a storage and synchronization layer and the pure functional behavior of the model.
  - The storage and synchronization layer stores the data of write transactions and returns the data in case of read transactions.
  - The behavior models the algorithm or state machine of the component. The behavior is triggered when certain memories or registers in the storage layer are accessed.

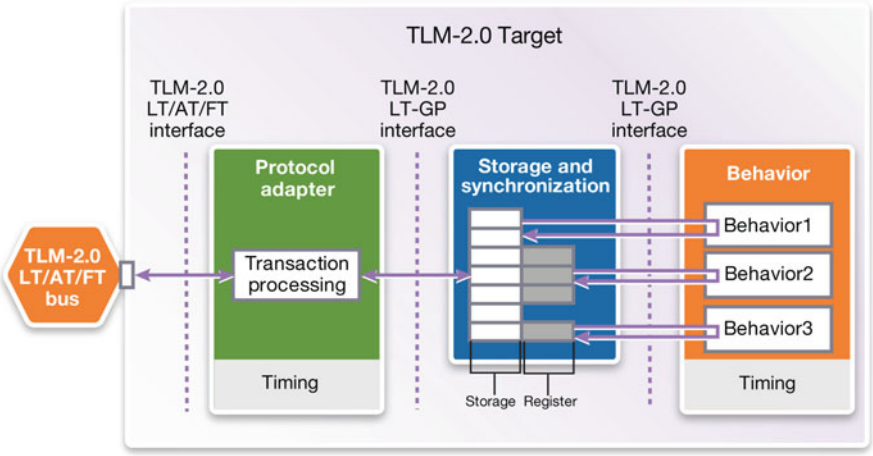


Fig. 34.5 SCML-based modeling pattern for target peripherals

- Finally, the different needs in timing accuracy can be addressed by separating the code that models the timing of the component from the pure functional behavior. SCML supports this separation by providing modeling objects for each of these layers.
  - The adaptation layer handles communication-related and data-independent timing aspects, e.g., the duration of a protocol phase or the number of outstanding transactions.
  - The behavior layer handles processing-related and data-dependent timing aspects.

The SCML modeling library greatly helps to reduce the modeling effort. In the context of commercial virtual prototyping projects, the effort for creating models can be further reduced by using model generation tools. For example, an SCML-based peripheral model can be automatically generated from an IP-XACT description of the register interface. Note that for the purpose of generating the register interface of a peripheral model, the IP-XACT importer only takes a subset of the IP-XACT standard into account, which is related to meta-data, register, and parameter information and which are supported by the SCML modeling objects.

34.2.3 System-Level Power Analysis

So far, the focus of this section has been on modeling functionality and timing, where the requirements are quite different depending on whether the VP is used for early architecture analysis or for software development. This section shows how to enable early power analysis irrespective of virtual prototyping use case.

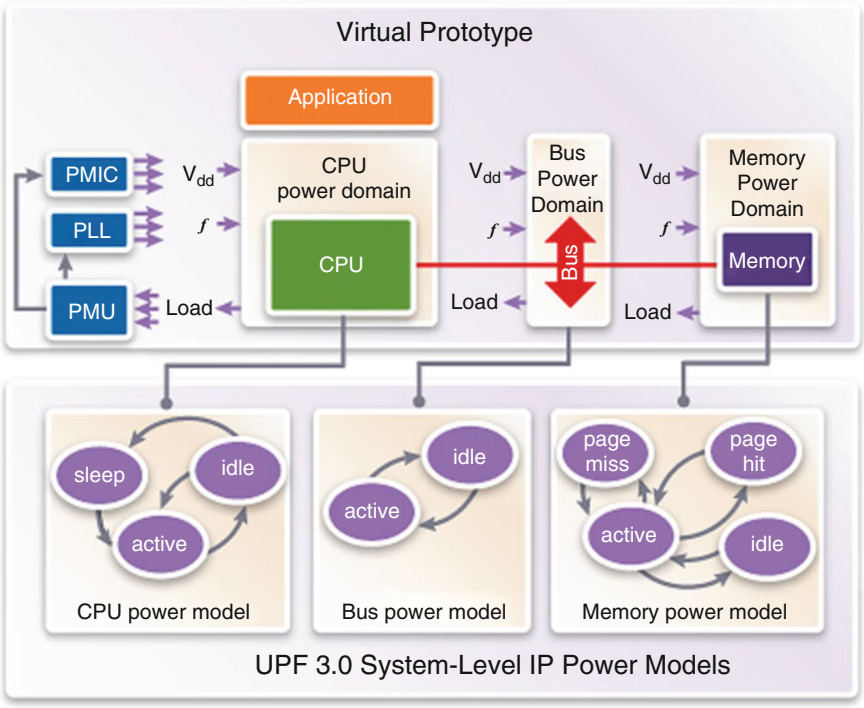


Fig. 34.6 Adding system-level power model as an overlay to VPs

The annotation of power information is performed through the use of system-level IP power models, which are power models of IP components specifically for use in system-level design. The format of system-level IP power models is defined by the IEEE 1801-2015 standard [11]. Originally the IEEE 1801 Unified Power Format (UPF) was defined to capture power intent for hardware implementation and verification. UPF is a format based on the Tool Command Language (TCL) [28] and defines the power supply and low power details as an overlay to the actual Hardware Description Language (HDL) implementation [10]. The new System-Level Power (SLP) features of the 1801-2015 "UPF-3.0" release extend the UPF TCL syntax to model power consumption as an overlay to a "host." Figure 34.6 shows an example of a VP with a UPF-3.0 system-level power overlay model. In this context, the UPF-3.0 power model calculates power consumption by observing the dynamic activity in the VP.

34.2.3.1 UPF-3.0 System-Level IP Power Models

A context-independent UPF-3.0 system-level power model comprises the following aspects:

- A set of buildtime and run-time parameters, which influence the power consumption of the respective IP. Examples of buildtime parameters are technology, size of a memory, or number of CPU cores. Examples of run-time parameters are voltage, frequency, or temperature.
- The set of relevant power states of the IP. These are not necessarily identical with the power supply states (e.g., `off`, `sleep`, `active`) but can also refer to operating modes with distinct power consumption signatures, e.g., a CPU in `WFI` (Wait For Interrupt) state or a video IP in `encode` or `decode` state.
- A set of functions to calculate the power consumption of the IP in the respective state and based on the set of parameters. The power functions need to separately return static and dynamic power consumption.
- The set of legal and illegal state transitions.

Now that the IEEE standard is ratified, it is expected that system-level IP power models are developed and distributed by IP teams (whether they be IP vendors or IP implementation teams within larger platform development groups).

34.2.3.2 UPF-3.0 System-Level Power Example

The example of a simple CPU power model in Fig. 34.7 illustrates the basic concepts of a UPF-3.0 power model:

- The power model is defined inside the `begin_power_model` and `end_power_model` commands.
- The `create_power_domain` command allows to represent hierarchy inside the power model, e.g., to represent different power states for the core and the cache inside one power model.

```
begin_power_model CPU
create_power_domain CORE -elements {}

add_parameter vdd                       -type runtime -default 1V          -description "voltage"
add_parameter frequency                 -type runtime -default 1000MHz     -description "frequency"
add_parameter temperature               -type runtime -default 25.0C       -description "temperature"
add_parameter percent_active_power_in_wfi    -type buildtime -default 10   -description ""
add_parameter mA_per_MHz_at_1V               -type buildtime -default 0.3  -description ""
add_parameter I_lkg_in_mA_at_1V_at_25C       -type buildtime -default 10mA -description ""

add_power_state -domain CORE \
    -state { OFF -power_expr {0mW 0mW} } \
    -state { WFI -power_expr { wfi_power {vdd frequency temperature} } } \
    -state { ACTIVE -power_expr {active_power {vdd frequency temperature} } }

add_state_transition -domain CORE \
     -transition {to_wfi      -from {ACTIVE OFF    -to WFI             -legal} \
     -transition {from_wfi    -from {WFI           -to {ACTIVE OFF}    -legal} \
     -transition {off         -from WFI            -to OFF             -legal} \
     -transition {on          -from OFF            -to WFI             -legal} \
     -transition {illegal1    -from OFF            -to ACTIVE          -illegal} \
     -transition {illegal2    -from ACTIVE         -to OFF             -illegal}
end_power_model
```

Fig. 34.7 Example of context-independent UPF-3.0 system-level power model

- The `add_parameter` command in the CPU power model defines three static buildtime parameters and three dynamic run-time parameters.
- The `add_power_state` command in the CPU power model defines three power states. The power consumption for the state `OFF` state is zero. For the other two states, the power consumption is calculated by the respective `wfi_power` and `active_power` functions. Both functions are sensitive to all of the three run-time parameters. This implies that the power functions need to be reevaluated whenever any of the run-time parameters in the sensitivity list changes.
- The `add_state_transition` command defines the set of legal and illegal state transitions.

The actual power functions are implemented outside of the scope of the actual power model and therefore not shown in this example.

In a second step, this context-independent power model can be instantiated in the context of a VP, which contains a TLM of a CPU. The corresponding integration layer of the CPU power model is depicted in Fig. 34.8

- The `apply_power_model` UPF command maps the run-time parameters in the CPU power model to corresponding signals of the CPU TLM.
- The same command is used to initialize the buildtime parameters in the CPU power model with static configuration parameters of the CPU TLM.
- The `add_edge_expression` command defines the conditions that cause a state transition in the power model.

In this simplistic example, the power model is triggered from a signal port of the TLM. This illustrates that the concept of UPF-3.0 IP power model is not limited to a system model but can be also applied to a RTL or gate-level representation of a component running in an HDL simulation, emulation, or FPGA prototyping environment. On the other hand, in a virtual prototyping environment, the transition of a power state can be triggered from all kinds of *observable events* in the TLM, e.g., the start or end of a transaction, a register access, an event on an analysis

```
::apply_power_model CPU -elements CPU0 -parameters \
  {  {vdd                          CORE0.VDD{V} } \
     {frequency                    CORE0.Frequency{MHz}} \
     {temperature                  CORE0.Temperature{C}} \
     {percent_active_power_in_wfi  /Power/percent_active_power_in_wfi} \
     {mA_per_MHz_at_1V             /Power/mA_per_MHz_at_1V} \
     {I_lkg_in_mA_at_1V_at_25C     /Power/I_lkg_in_mA_at_1V_at_25C{mA}} \
  }

::snps_upf::add_edge_expression -elements CPU0 -domain CORE \
    -state {OFF     -edge_expr {CORE0.VDD == 0} } \
    -state {WFI     -edge_expr {CORE0.wfi == 1} } \
    -state {ACTIVE  -edge_expr {CORE0.wfi == 0} }
```

Fig. 34.8 Example of context-dependent UPF-3.0 integration layer

instrumentation point, a specific software symbol executed on an Instruction-Set Simulator (ISS), etc. Due to the tool-specific nature of the edge expression, this command is so far not part of the official UPF standard and therefore implemented with a Synopsys-specific command.

Taken together, the context-independent power model and the context-dependent integration layer create a power analysis overlay model of a VP as depicted in Fig. 34.6.

34.2.3.3 Accuracy Considerations

The goal of system-level power analysis is not to provide 100% accurate power measurements but to replace high-level power estimation currently done with static spreadsheets. The actual accuracy of system-level power analysis depends mainly on the granularity of the power model and on the characterization of the power functions:

- The granularity of a SLP model is determined by the level of detail in the power model. For example, a CPU power model can be modeled as
  - a simple monolithic state machine as shown in the example above.
  - multiple domains for cores, cache, coprocessor, etc., each with their own state machine.
  - a detailed instruction-level power model, which calculates power based on the specific energy of each executed instruction.
- The characterization determines how the power expressions calculate the power consumption based on power estimates and/or measurements.
  - An early power characterization can be defined using high-level estimates, e.g., based on the extrapolation of power measurements from previous projects.
  - Once RTL and technology libraries are available, RTL or gate-level power estimation tools can be used to generate look-up tables, which determine power consumption based on design parameter configuration and operating mode.
  - Post-silicon power measurements can still be valuable to characterize the power consumption of a reusable IP block for usage in subsequent projects.

Despite the high level of abstraction, it turns out that VPs with system-level power analysis models provide power estimates in the order of 85–90% accuracy, which are good enough to steer architecture design decision and to guide software development in the right direction [8, 22].

34.2.4 Summary

This section provided an overview of the modeling methodologies enabling virtual prototyping. The first part surveyed the IEEE 1666 SystemC Transaction-Level Model (TLM) standard, emphasizing the Loosely Timed (LT) and Approximately

Timed (AT) modeling styles and their respective extensions. The second part gave an introduction to the new IEEE 1801-2015 UPF-3.0 modeling standard for system-level power analysis. The subsequent section elaborates on how these modeling techniques are applied to the creation and usage of VPs for early architecture analysis.

34.3 Virtual Prototyping for Architecture Design

Incorporating more and more functions and features into electronic products directly translates into increasing SoC design complexity. Devices integrate a multitude of heterogeneous programmable cores to achieve the necessary flexibility and power efficiency. The diverse communication requirements of all these cores lead to a complex interconnect and memory infrastructure to provide the required storage and communication bandwidth. For this purpose, the SoC interconnect and memory subsystem feature complex mechanisms like distributed memory, cascaded arbitration, and Quality of Service (QoS). As a result, dimensioning the SoC architecture, and in particular the interconnect and memory infrastructure, poses a variety of formidable design challenges:

Large Design Space
: Due to the complexity and configurability of the SoC infrastructure IP (interconnect, memory), tailoring the SoC infrastructure to the specific needs of the product requirements is a nontrivial task.

Dynamic Workload
: Multiple applications running at different points in time are sharing a limited set of available resources. Hence, the workload on the SoC architecture is difficult to estimate due to the multitude of product use cases.

High Price of Failure
: A weakly dimensioned SoC architecture leads to insufficient product performance (under-design) or excessive cost and power consumption (over-design). Both cases hamper the market opportunity of the final product.

High Potential for Optimization
: All the design decisions, which impact power, performance, and cost in a big way, need to be taken at the beginning of the development process.

This section first provides an introduction to virtual prototyping for architecture design and then dives into more detail about specific methods for architecture exploration, optimization, and validation.

34.3.1 Introduction

The introduction first discusses traditional methods for architecture design and then introduces a state of the art flow and modeling methodology based on a commercial virtual prototyping solution.

34.3.1.1 Traditional Methods

Architecture definition has always been a necessary step in any SoC design project. Traditionally, the performance has been analyzed using spreadsheets or detailed hardware simulation. However, the design complexity has reached a level where these methods are not appropriate anymore. On the one hand, static spreadsheet analysis does not take the dynamic behavior of multiple software applications and multiple levels of scheduling and arbitration in the executing hardware platform into account. This bears a great risk of mis-predicting the actual performance, which can lead to under- or over- design of the system architecture. On the other hand, hardware simulations are available late in the cycle, run very slow, and do not provide system-level performance analysis results. Hence, this is also not a suitable approach for early architecture analysis and optimization.

34.3.1.2 Virtual Prototyping Flow for Early Architecture Analysis

Synopsys Platform Architect for Multi-Core Optimization (MCO) is a virtual prototyping environment for early and accurate system-level performance analysis. This comprises libraries of simulation models for all relevant SoC components as well as tools for the assembly, simulation, and analysis of complete SoC platforms. A typical setup for memory subsystem performance analysis and optimization is shown in Fig. 34.9 below. The following paragraphs briefly describe the key elements in the model library and the architecture analysis flow. Please refer to [15] for a more complete introduction.

The flow to systematically analyze and optimize the architecture in Platform Architect is depicted in Fig. 34.9:

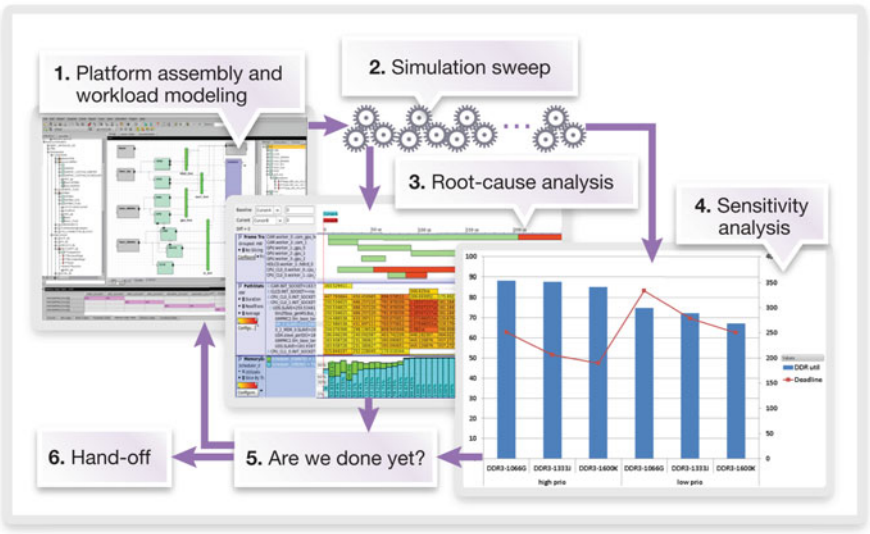


Fig. 34.9 Architecture analysis and optimization flow

1. Platform Assembly and Workload Modeling
   In the first step, the SoC performance model is assembled, connected, and configured in the Platform Architect authoring environment. This is quickly done based on the available libraries for workload, interconnect, and memory subsystem models.
2. Simulation Sweep
   Platform Architect generates a SystemC-based simulation of the SoC performance model. The simulation records a large variety of power and performance metrics into an analysis data base.
3. Performance Analysis
   The recorded data can be visualized and post-processed in the Platform Architect analysis tool. Various charts for throughput, latency, utilization, and contention allow the identification of performance issues. We can zoom on the time axis and further slice the results into the contribution from individual components for detailed root cause analysis.
4. Sensitivity Analysis
   Apart from single simulation runs, Platform Architect can also generate parameter sweeps, where a set of simulations is executed with user-defined configuration scenarios. This allows to systematically analyze the impact of the selected design parameters on high-level performance metrics. Each simulation result can be analyzed individually, but the results from all simulations are also aggregated into pivot chart tables for further post-processing in spreadsheet tools, e.g., Excel.
5. Are we done yet?
   Based on the analysis results, the architect needs to decide if the performance requirements in terms of throughput and latency cost are met. If this is confirmed, there might still be potential to improve utilization or to reduce contention in order to further optimize headroom, cost, or power consumption. In those cases, the iterative optimization loop continues by further modifying configuration parameters and setting up simulation sweeps until the design goals are reached.
6. Hand-off
   At the end of the optimization process, the final design configuration is handed off to the implementation team. As soon as the implementation becomes available, it can replace the system-level performance model with the RTL model. This allows the validation of the analysis results with the highest possible accuracy.

This generic iterative exploration and optimization flow can be applied to all kinds of architecture design problems. The next paragraph elaborates on how the architecture model should be constructed, depending on the specific objective of the architecture design project.

34.3.1.3 Modeling Methodologies for Early Architecture Analysis

The dimensioning of the SoC architecture is the first step in a design project. As depicted in Fig. 34.10, the input comes from the marketing requirements in terms of required features, supported features, performance numbers, and product cost.

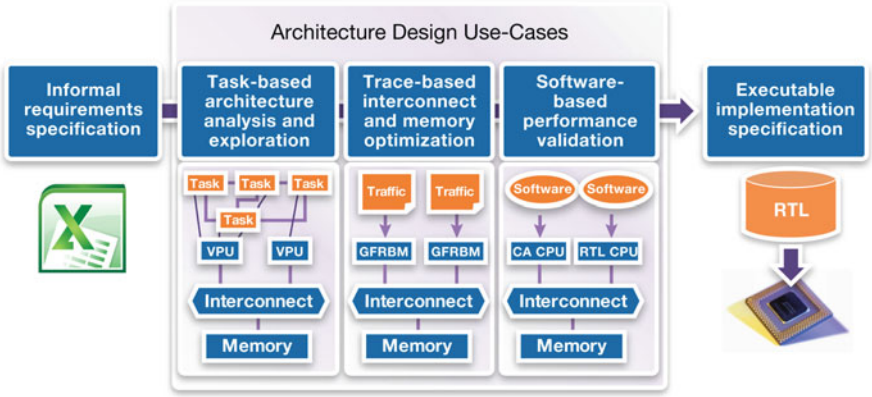


Fig. 34.10 Modeling methodology for different architecture design use cases

The outcome of the architecture design process is a detailed specification of the implementation. In between, three differentiated use cases have emerged over the last two decades of commercial deployment of architecture design methodologies.

All three steps include the modeling of the interconnect and memory architecture, because in many cases, this is the primary objective of the architecture design project. This is because individual IP blocks and subsystems can be developed independently, but when they are integrated into the SoC platform, the interconnect and memory subsystems need to satisfy the accumulated requirements from all IPs. Hence, the analysis of the interconnect and memory performance is typically the "common denominator" of all architecture design use cases.

The fundamental difference between the architecture design use cases is in how to model the remaining IP blocks and subsystems in the SoC:

- For the purpose of architecture validation, the actual software is executed on a fully functional and timing accurate VP, which is similar in nature to a VP for early software development: Instruction-Set Simulators (ISSs) are used for all programmable components, and functional and bit-accurate peripheral models are used for the nonprogrammable components. In addition, all models need to be enhanced with timing, which greatly increases the modeling effort and impacts the simulation speed.
- For the purpose of early architecture analysis and interconnect/memory optimization, the IP blocks are represented as abstract *workload models*. These task-based or trace-based workload models are nonfunctional and only represent the processing and communication requirements of each IP component, irrespective of whether they are programmable or not. The nonfunctional modeling of workloads is the key concept to achieve the necessary flexibility, modeling productivity, and simulation speed with sufficient temporal accuracy for early architecture analysis and exploration.

The following sections discuss the different architecture design use cases in more detail.

34.3.2 Software-Based Performance Validation

Historically, performance validation was the first commercially deployed architecture design use case. The idea is to build a functional and cycle-accurate VP of the complete SoC. The programmable IP subsystems require cycle-accurate representation of the CPU, capable of running the actual software. Depending on availability, this can be a Cycle Accurate (CA) SystemC TLM Instruction-Set Simulator (ISS) or the RTL of the CPU running in cosimulation or coemulation mode with the SystemC TLM platform. The nonprogrammable IP blocks need to be modeled in terms of CA SystemC TLMs. The benefit of such a fully accurate VP is that it allows the early validation of the final SoC performance. The analysis visibility into hardware and software enables the identification of performance issues and tuning of design and configuration parameters to optimize performance.

However, the overall return of investment (RoI) has proven to be challenging, especially for the growing complexity of complex many-core SoC platforms:

- Creating such a fully functional and cycle-accurate platform model requires a lot of initial modeling effort. Even if all models are available, it can be cumbersome to configure the software such that all the relevant traffic scenarios are covered.
- The simulation speed and the turnaround time for any change to the hardware or software is very slow.

For these reasons, performance validation is typically done using emulation or hardware prototyping methods. A new trend is hybrid emulation and prototyping, which allows to combine the best of both worlds:

- Leverage emulation and hardware prototyping for large IP blocks, e.g., CPU, GPU, and custom IP, to achieve reasonable simulation speed and avoid the effort to create cycle-accurate models.
- Leverage virtual prototyping for interconnect and memory subsystem to analyze and optimize performance critical parameters with high analysis visibility and fast turnaround time.

Architecture analysis with VPs starts very early in the development process. At this point, neither the software nor the RTL of the major IP blocks is available. Therefore, initial performance analysis is carried out using workload models instead of real software. As the platform specification and implementation matures, the workload models can be incrementally replaced by the cycle-accurate functional models or the RTL. This gradually converts the architecture exploration model on the left side of Fig. 34.10 into a cycle-accurate VP as depicted on the right. This way, the initial assumptions in the workload model can be validated, and the architecture can be fine tuned.

The following sections describe in more detail the creation of VPs for architecture analysis using trace-based and task-based workload modeling.

34.3.3 Trace-Based Interconnect and Memory Optimization

In many cases, the interconnect and memory optimization is the key concern of the architecture definition phase. For this purpose, it is often sufficient to model the workload of all relevant initiator components in terms of transaction trace files, which are replayed by Generic File Reader Bus Masters (GFRBMs). Trace-based workload modeling has proven to be the most productive methodology to quickly and accurately analyze and optimize this critical part of the SoC architecture. Traces are available early in the development process, long before the actual software exists. They can be easily recorded or generated, they execute fast, and they represent the transaction sequence and timing of the real application with sufficient accuracy.

34.3.3.1 Traffic Generation

A GFRBM is sufficient to model the traffic generated by all kinds of initiator components like CPU, GPU, DMA, etc. The critical portion is the creation of elastic traces, which accurately represent the actual traffic of the corresponding initiator component. Here elasticity refers to the requirement, that the trace generator needs to respond to a change in the interconnect and memory architecture in the same way as the corresponding initiator component. An important aspect of elasticity is the synchronization of different traffic flows: If one traffic flow is triggered by another flow, then this dependency needs to be explicitly represented in the trace-based workload model. An example of such an elastic trace is shown in Fig. 34.11.

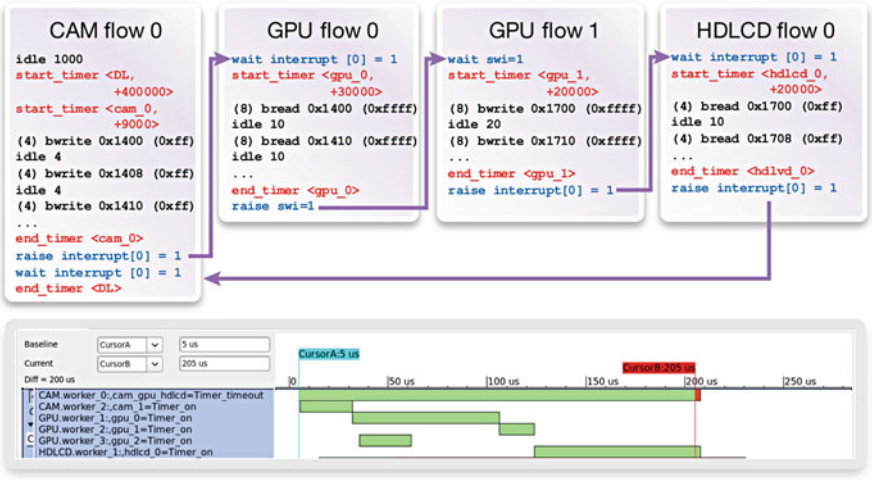


Fig. 34.11 Elastic trace-based workload model (*top*) with deadline analysis (*bottom*)

The upper part of Fig. 34.11 shows an example of the transaction-level trace format that is executed by the GFRBM. Each traffic flow is described as a sequence of reads and writes with the relevant transaction attributes, e.g., burst size, command, address, byte enables, etc. As opposed to using absolute time stamps, the idle command defines the relative number of cycles between two subsequent transactions.

Multiple traffic flows can be synchronized by using the raise and wait for internal and external interrupt signals. In the example above, flow 0 on the GPU only starts after flow 0 on the camera raises interrupt 0. In the same way, flow 1 on the GPU waits for flow 0 to finish.

It is also important to capture performance constraints of traffic flows. For example, the `cam_0` timer in Fig. 34.11 expires after 9000 cycles to indicate that the transaction sequence should be processed before this deadline. The lower part of Fig. 34.11 shows the visualization of the deadlines: The end-to-end constraint of the camera, GPU, and HD-LCD starts at 5 µs and turns red at 205 µs, indicating that the 200 µs deadline is just missed.

34.3.3.2 Transaction-Level Models for Interconnect and Memory Subsystem

Obviously the key ingredient for analyzing and optimizing the performance of the interconnect and memory subsystem are sufficiently accurate models of these components. Commercially available virtual prototyping environments provide libraries of SystemC TLMs at different levels of abstraction.

- Highly configurable approximately timed models, which can be used to mimic a specific IP.
- Highly accurate model of a specific IP, which represent all relevant design and configuration parameters

An example of a configurable approximately timed model is the generic Multi-Port Memory Controller provided in the Synopsys Platform Architect model library. This memory controller model is based on the Fast Timed (FT) TLM protocol to support multiple bus protocols (see paragraph on Extended AT in Sect. 34.2.1). It incorporates the features of modern memory controllers, e.g., transaction re-ordering, address mapping, configurable number of ports, buffer sizes, frequency ratio, and QoS. This also includes a timing model of all commonly used Double Data Rate (DDR) standards (DDR2, DDR3, DDR4, mDDR, LPDDR2, LPDDR3, LPDDR4, DDR3-3DS) with their respective speed bins and device types [16]. An example of an accurate IP-specific model for architecture analysis is the Architects View (AV) TLM model of the FlexNoC interconnect from Arteris [19]. The port interfaces of the AV FlexNoC model use an extended TLM-2.0 AT protocol, which accurately represents the FlexNoC NTTP protocol with extended attributes and phases, but which is also compliant with the AT-BP (see Sect. 34.2.1). Internally the AV FlexNoC model represents all the performance-relevant aspects of the NoC architecture, like the topology, serialization, clock schemes, buffering, arbitration

schemes, pipeline stages, and transaction contexts. This also includes advanced QoS schemes like run-time bandwidth regulation. The FlexNoC model is generated from the NoC configuration tool, which is later used to generate the actual RTL implementation. This way, the optimized interconnect configuration is seamlessly used for implementation. The model library from Synopsys Platform Architect contains similar models for other popular interconnect IP, e.g., ARM CoreLink NIC-400.

Based on these kinds of available TLMs for architecture analysis, it is very little effort to assemble a performance model of any SoC platform, which allows to analyze and optimize the memory subsystem [17, 20, 24].

34.3.4 Task-Based Architecture Analysis and Exploration

The most recent commercially deployed architecture design methodology is the early exploration and optimization of complex Multi-Processor System-on-Chip (MPSoC) platforms using task-based workload models. This allows the quantitative analysis of performance and power metrics to avoid SoC market failure due to underperforming or overly power hungry architectures. The key architecture questions that SoC hardware architects can analyze are:

- How to partition the application into fixed HW accelerators and software executing on processors?
- What is the optimal number and type of CPUs, GPUs, Digital Signal Processors (DSPs), and HW accelerators?
- How to dimension the interconnect and memory architecture?
- What is the expected performance/power curve?

34.3.4.1 Modeling Methodology

As depicted in Fig. 34.12, the modeling methodology for task-based architecture analysis follows the Polis Y-chart approach [5], similar to the framework described in the ▸ Chap. 9, “Scenario-Based Design Space Exploration”.

- SoC application workloads such as CPU load, imaging, video encoding and decoding, modem, and network packet processing are represented as an application task graph.
- The VP of the SoC platform contains all relevant processing elements as well as interconnect and memory resources. The key component is the Virtual Processing Unit (VPU), which represents all kinds processing elements such as CPUs, GPUs, DSPs, and HW accelerators. VPUs are high-level processor models that execute the portion of the task graph [13, 18].
- The task-based application workload model is mapped to the architecture model to construct an executable specification of the application running on the hardware platform.

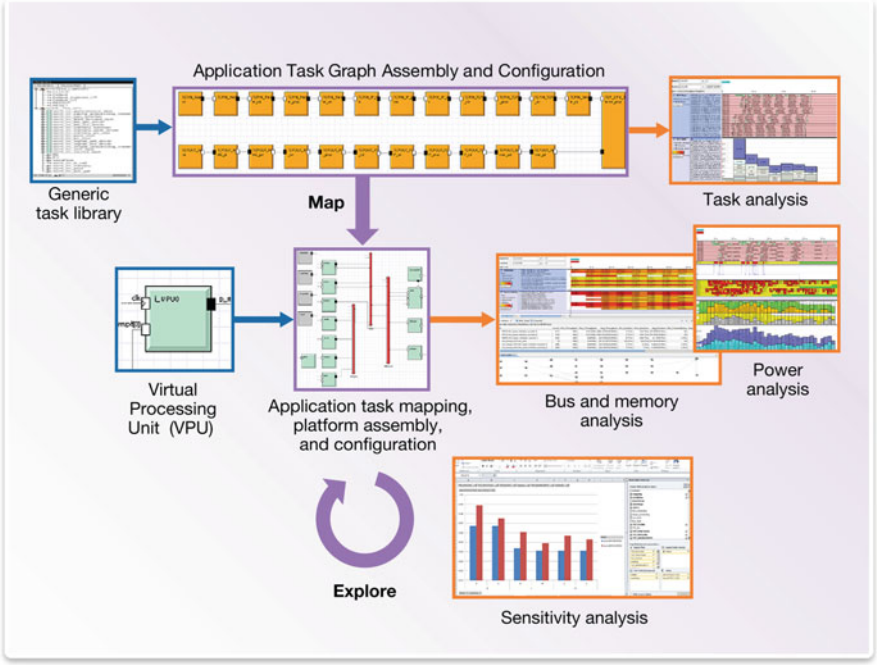


Fig. 34.12 Early architecture analysis with task-based workload models

The following sections describe the different aspects of application modeling, platform modeling, and mapping in more detail.

34.3.4.2 Task-Based Workload Models

In general, a task-based workload model captures the processing and communication requirements of the application. As showcased in Fig. 34.13, the overall application is broken down into a set of tasks, which exhibits the available coarse-grained parallelism. The connections in the task graph denote the execution precedence, e.g., in this example Task C and D execute after B. In addition, each task is characterized with a set of processing- and communication-related parameters. A typical set of parameters is depicted in Fig. 34.13:

- A source task like Task A, the `wait_cycles` parameter specifies the delay between two consecutive activations
- The `processing_cycles` specifies minimum number of cycles for which a task occupies a resource.
- The `load_ratio` `store_ratio` specify the additional communication overhead, e.g., on average Task B generates 50 load and store transactions per activation.

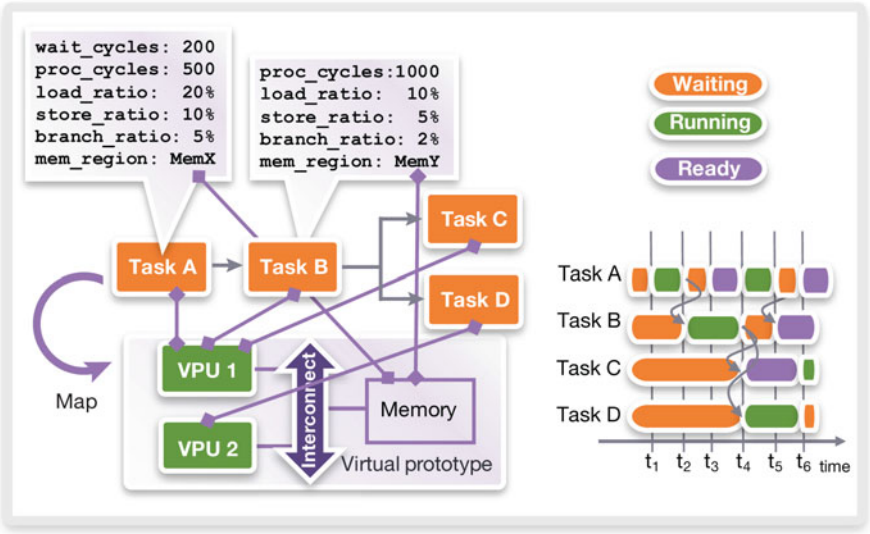


Fig. 34.13 Specification, mapping, and execution of an application task graph

- The `branch_ratio` can be used to model jumps in the address sequence and additional processing overhead for mis-predicted branches.
- The `mem_region` defines a logical name for the memory location.

In the mapping step, tasks are assigned to VPUs, which represent the execution resources, and the logical memory regions assigned to physical memories in the platform.

From the modeling perspective, tasks and connections are realized on top of standard SystemC concepts like threads and events. In addition, tasks have explicit states like *Created*, *Ready*, *Running*, *Waiting*, *Suspended*, and the ability to *consume* processing time. This enables the modeling of software processes executing in the context of an Operating System (OS). The task state trace on the right side of Fig. 34.13 showcases the execution of the given example task graph:

- In the beginning, all task are in state *Waiting*
- At t_1, the wait cycles of source Task A are expired. VPU 1 is available, so Task A changes immediately into state *Running*.
- At t_2, the processing cycles of Task A are finished. Task A triggers Task B, which changes into state *Running*.
- At t_3, the wait cycles of Task A are again expired, but this time VPU 1 is still occupied with Task B, so Task A transitions into state *Ready*.
- At t_4, Task B is done, so Task A becomes *Running* on VPU 1. Also, Task B triggers Task C and D. C is blocked by Task A on VPU 1 and remains *Ready*, but Task D can immediately transition into state *Running* on VPU 2.

In reality, more complex scheduling algorithms with priorities, preemption, time slices, etc., influence the execution pattern. The actual duration a task remains in state *Running* is further impacted by dynamic effects like bus arbitration and dynamic memory latencies.

Synopsys provides a generic task library with a set of configuration parameters, so users can rapidly compose a task graph without manual modeling effort. This library provides a set of generic configurable tasks to create a nonfunctional performance model of arbitrary application topology.

For data processing applications, such as audio, video, networking, or wireless communications, it is typically straightforward to define the application task graph using the elements in the generic task library. For control-oriented applications, e.g., in the automotive domain, and for higher-level applications running on top of an OS, the application task graph can be automatically generated from software execution traces.

34.3.4.3 Performance Model of the System-on-Chip

Multi-core architectures are composed of SystemC TLMs, such as interconnect, memory controller, DMAs, and other components which are available in the Platform Architect model library. The VPU models the processing elements (CPUs, GPUs, DSPs, and HW accelerators), which can execute a task graph, or a portion of the task graph. The VPU task scheduler supports preemption and time slicing of tasks for modeling of interrupts and arbitrary OS scheduling algorithms. The provided set of default scheduling algorithms can be extended by the user. The VPU also comes with a library of components for traffic generation, cache modeling, inter-VPU communication, and interrupt handling to model the realistic execution of a task graph with sufficient accuracy.

34.3.4.4 Application to Architecture Mapping

The next step is to map the application task graph onto the VPUs. This way the tasks are assigned to a physical execution resource, and the logical memory regions in the task graph are mapped to physical memories in the platform. The number of processing resources per VPU is configurable and determines how many tasks can run in parallel. For example, a VPU with four resources represents a Symmetric Multi-Processing (SMP) cluster with four cores. VPUs with different parameters (scheduling algorithm, clock period, traffic generation, etc.) represent an asynchronous multiprocessor (AMP) subsystem. A VPU with only one task mapped to it can represent a dedicated hardware block.

The result is an executable system performance model, which simulates the execution of a task-based workload model on a resource constraint platform. The analysis monitors measure a variety of performance metrics like latency, throughput, utilization, and contention. This way, an SoC architect can identify performance issues, bottlenecks, or underutilization of resources [8].

34.3.4.5 Joint Power and Performance Analysis

The availability of the IEEE 1801-2015 UPF-3.0 standard for system-level power analysis [11] enables the early estimation of the system power consumption by adding power monitors as an overlay to a VP for performance analysis; see Sect. 34.2.3 and Fig. 34.6). Compared to static power analysis based on spreadsheet, this provides much more realistic power estimates, because the dynamic activity of each component in the platform is taken into account. This way, architects can analyze the impact of architecture design decision on the power consumption [9]. Many state-of-the-art SoC platforms optimize the power consumption using run-time power management schemes, e.g., clock gating, power gating, and Dynamic Voltage and Frequency Scaling (DVFS) [10]. These power management schemes bear great potential to reduce power consumption, but they also come at additional cost. Hence power management adds another dimension to the architecture design space. Unfortunately, power consumption and performance cannot be considered in isolation. For example, reducing voltage and frequency reduces power but increases execution time. The resulting impact on the energy consumption is not obvious, so without quantitative analysis, it is difficult to decide between power management strategies like Run Fast Then Stop (RFTS) or DVFS.

As depicted in Fig. 34.14, adding a model of the power management to a VP for architecture analysis enables the quantitative analysis of power management strategies, including the impact of power management on power and performance:

- How should the SoC be partitioned into DVFS domains to best serve the target application use cases?
- Based on the activity profile of a component, do the power savings justify the additional cost for applying clock gating or power gating or both?

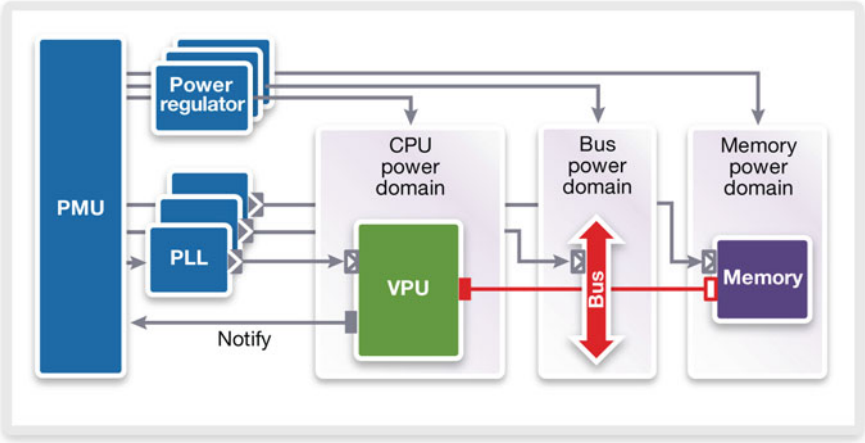


Fig. 34.14 DVFS modeling for joint power and performance analysis

- How many DVFS operating points are needed to effectively reduce power consumption?
- How aggressive can the frequency be reduced before application violates real-time requirements?

Figure 34.14 shows how to model the impact of DVFS and power management in the context of a multi-core platform in Platform Architect for MCO. A functional model of the Power Management Unit (PMU) is part of the SoC platform model to take the performance and power aspect into account. The processing elements (the VPUs) notify the PMU when they become active or idle. In response to this, the PMU model controls the frequency of the processing elements and the voltage levels of the power supply regulators. This captures the impact of the DVFS power management on the performance: The execution time of the tasks running on the VPU depends on the actual frequency. The same task takes longer when the clock is running at a lower frequency. The frequency and voltage levels are also used as run-time parameters in the UPF-3.0 power models to measure actual power consumption.

The outcome of the early analysis of power and performance is an optimized power architecture:

- An optimized specification of the system-level power intent, including the power supply architecture and the grouping of the SoC into power domains.
- The definition of the most promising power management policies.
- A realistic estimation of the system-level power and energy consumption for a given workload.

The UPF standard can also be used to export the resulting optimized system-level power intent from the virtual prototyping environment to the subsequent implementation and verification tools.

34.4 Conclusions

This section provided an overview of virtual prototyping for architecture design. It introduced software-based performance validation, trace-based interconnect/memory optimization, and task-based architecture exploration as the three main architecture design use cases, which are in commercial deployment today. Especially the joint power and performance analysis based on task-based application workload models sees growing adoption, because it is the most effective approach to cope with the "complexity wall" [26] under the given competitive landscape and time-to-market pressures.

Acknowledgments The author acknowledges Tom De Schutter for contributing the sections on software development and on system validation to the introduction of this chapter as well as Alan Gibbons for contributing the section on system level power analysis to the introduction of this chapter.

References

1. AMBA AXI and ACE Protocol Specification (2013). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html
2. AMBA-PV Extensions to TLM Developer Guide (2015). http://infocenter.arm.com/help/topic/com.arm.doc.dui0846f/DUI0846F_ambapv_extensions_to_tlm_2-0_dg.pdf
3. ARM Fast Models. http://www.arm.com/products/tools/models/fast-models
4. Bailey B et al (eds) (2010) TLM-driven design and verification methodology. Cadence Design Systems, San Jose. https://www.synopsys.com/vpbook
5. Balarin F et al (1997) Hardware-software co-design of embedded systems: the POLIS approach. Kluwer Academic Publishers, Boston
6. Barroso L, Holzle U (2007) The case for energy-proportional computing. Computer 40(12): 33–37. doi: 10.1109/MC.2007.443
7. Datta S, Bonnet C, Nikaein N (2012) Android power management: current and future trends. In: 2012 first IEEE workshop on enabling technologies for smartphone and internet of things (ETSIoT), pp 48–53. doi: 10.1109/ETSIoT.2012.6311253
8. Fadi Aboud IC (2014) Balancing power performance and user experience using virtual prototyping. In: Proceedings of the synopsys users group conference (SNUG), Israel. https://www.synopsys.com/news/pubs/snug/2014/israel/B2_Aboud_pres_user.pdf
9. Fischer B, Cech C, Muhr H (2014) Power modeling and analysis in early design phases. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), pp 1–6. doi: 10.7873/DATE.2014.210
10. Flynn D, Aitken R, Gibbons A, Shi K (2007) Low power methodology manual for system-on-chip design. Springer, New York
11. IEEE Standard for Design and Verification of Low-Power Integrated Circuits (2015). http://standards.ieee.org/getieee/1801/download/1801-2015.pdf
12. IEEE Standard for Standard SystemC Language Reference Manual (2011). http://standards.ieee.org/getieee/1666/download/1666-2011.pdf
13. Kempf T, Doerper M, Leupers R, Ascheid G, Meyr H, Kogel T, Vanthournout B (2005) A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms. In: Proceedings of design, automation and test in Europe, vol 2, pp 876–881. doi: 10.1109/DATE.2005.21
14. Kogel T (2006) Peripheral modeling for platform driven ESL design. In: Burton M, Morawiec A (eds) Platform based design at the electronic system level. Springer, Dordrecht, pp 71–85
15. Kogel T (2013) Designing the right architecture, SoC interconnect and memory optimization with synopsys platform architect. Synopsys whitepaper. https://www.synopsys.com/cgi-bin/proto/pdfdla/pdfr1.cgi?file=pa_soc_v4_wp.pdf
16. Kogel T (2016) Optimizing DDR memory subsystem efficiency, Part 1: the unpredictable memory bottleneck. Synopsys whitepaper. https://www.synopsys.com/cgi-bin/proto/pdfdla/pdfr1.cgi?file=optimizing-ddr-efficiency-p1-wp.pdf
17. Kogel T (2016) Optimizing DDR memory subsystem efficiency, Part 2: case study. Synopsys whitepaper. https://www.synopsys.com/cgi-bin/proto/pdfdla/pdfr1.cgi?file=optimizing-ddr-efficiency-p2-wp.pdf
18. Kogel T et al (2005) Integrated system-level modeling of network-on-chip enabled multi-processor platforms. Springer, Dordrecht
19. Lecler J-J, Baillieu G (2011) Application driven network-on-chip architecture exploration & refinement for a complex SoC. Des Autom Embed Syst 15(2):133–158. doi: 10.1007/s10617-011-9075-5
20. Patel S, Sood B, Semiconductor F (2014) Quick, re-usable and cost effective approach to create accurate models using synopsys platform architect framework for early system level performance analysis. In: Proceedings of the synopsys users group conference (SNUG), India. http://www.synopsys.com/news/pubs/snug/2014/India/paper_sood.pdf

21. Reyes V (2012) Virtualized fault injection methods in the context of the ISO 26262 standard. SAE Int J Passenger Cars Electron Electr Syst 5(1):9–16
22. Schurmans S, Zhang D, Auras D, Leupers R, Ascheid G, Chen X, Wang L (2013) Creation of ESL power models for communication architectures using automatic calibration. In: 2013 50th ACM/EDAC/IEEE design automation conference (DAC), pp 1–6. doi: 10.1145/2463209.2488804
23. Schutter TD (ed) (2014) Better software. Faster! best practices in virtual prototyping. Synopsys Press. https://www.synopsys.com/vpbook
24. Skrzeszewski TK, Intel Corp. (2015) ATOM mobile SoC performance and power architecture exploration. In: Synopsys users group conference (SNUG), Santa Clara. http://www.synopsys.com/news/pubs/snug/2015/silicon-valley/mb08_skrzeszewski_paper.pdf
25. Synopsys DW TLM library. http://www.synopsys.com/Prototyping/VirtualPrototyping/VPModels/Pages/DW-TLM-Library.aspx
26. Teich J (2012) Hardware/software codesign: the past, the present, and predicting the future. Proc IEEE 100(Special Centennial Issue):1411–1430. doi: 10.1109/JPROC.2011.2182009
27. The SystemC Modeling Library (SCML). http://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi
28. Tool Command Language (TCL). http://www.tcl.tk

Part IX
Applications and Case Studies

Joint Computing and Electric Systems Optimization for Green Datacenters

35

Ali Pahlevan, Maurizio Rossi, Pablo G. Del Valle, Davide Brunelli, and David Atienza

Abstract

This chapter presents an optimization framework to manage green datacenters using multilevel energy reduction techniques in a joint approach. A green datacenter exploits renewable energy sources and active Uninterruptible Power Supply (UPS) units to reduce the energy intake from the grid while improving its Quality of Service (QoS). At server level, the state-of-the-art correlation-aware Virtual Machines (VMs) consolidation technique allows to maximize server's energy efficiency. At system level, heterogeneous Energy Storage Systems (ESS) replace standard UPSs, while a dedicated optimization strategy aims at maximizing the lifetime of the battery banks and to reduce the energy bill, considering the load of the servers. Results demonstrate, under different number of VMs in the system, up to 11.6% energy savings, 10.4% improvement of QoS compared to existing correlation-aware VM allocation schemes for datacenters and up to 96% electricity bill savings.

Acronyms

AC	Alternating Current
BFD	Best-Fit-Decreasing
CVMP	Correlation-aware VM Placement
CRAC	Computer Room Air Conditioning
CTI	Charge Transfer Interconnect
DC	Direct Current
DoD	Depth-of-Discharge

A. Pahlevan (✉) • P. G. Del Valle • D. Atienza
Embedded Systems Laboratory (ESL), EPFL, Lausanne, Switzerland
e-mail: ali.pahlevan@epfl.ch; pablo.garciadelvalle@epfl.ch; david.atienza@epfl.ch

M. Rossi • D. Brunelli
Department of Industrial Engineering, University of Trento, Trento, Italy
e-mail: maurizio.rossi@unitn.it; davide.brunelli@unitn.it

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_35

DP	Dynamic Programming
DSO	Distribution System Operator
DVFS	Dynamic Voltage and Frequency Scaling
ESS	Energy Storage Systems
HES	Hybrid Electric Systems
IT	Information Technology
MAPE	Mean Average Percentage Error
NOCT	Nominal Operating Cell Temperature
PCP	Peak Clustering-based Placement
PDU	Power Distribution Unit
PV	Photovoltaic
QoS	Quality of Service
SoC	State of Charge
SoH	State of Health
STC	Standard Test Conditions
UPS	Uninterruptible Power Supply
V/f	Voltage/Frequency
VM	Virtual Machine

Contents

35.1 Introduction

Ever-increasing demands for computing and growing number of clusters and servers in datacenters have ramped up the power consumption costs as an undesirable effect [20]. On the other hand, traditional fossil fuel concerns, carbon emissions, and global warming impose the introduction of more sustainable energy sources and behavioral change of people [41], since 10% of the global consumption of electrical energy has been estimated to be consumed by Information Technology (IT) infrastructures [14].

To optimize the operation of a datacenter, it is crucial to minimize both IT and cooling energy consumptions. Server consolidation [26] is one of the widely used techniques to reduce the energy overheads, which minimizes the number of active servers by packing workloads or virtual machines (VMs) into the minimal number of active servers exploiting a virtualized environment. Large virtualized datacenters use renewable energy to reduce their dependence on costly and brown energy from the grid [33].

In the recent years, all the big energy consumers in the IT market (Amazon, Google, Rackspace, etc.) have already introduced renewable energy sources in their supply chain, locating their infrastructures in suitable geographical locations around the world. The penetration of renewable and green energy sources is almost none for company-owned datacenters, IT infrastructures located in the same corporate building where the business is run, mostly in urban environments.

Solar energy is the most effective renewable source employed in green datacenters since Photovoltaic (PV) modules can be easily located close by the datacenter and the converted energy can be immediately used without distribution. Moreover it is the most suitable for small to medium datacenters (up to few hundreds kWs of IT power) located in urban environments where wind turbines and water storage infrastructures may not be built, given the space required for such infrastructures.

Renewable energy sources are not constant over the time; their intensity depends on weather, geographical position of the plant, and seasons; moreover a maximum in the energy intake rarely corresponds with a maximum in the demand. However, estimating their short-term trend (one day ahead) with small error (Mean Average Percentage Error (MAPE) close to 10%) is possible, as it has been demonstrated in [11]. Similar results can be expected when dealing with electricity demand prediction at building scale (few tens of kWs) [31]. To tackle the imbalance between energy intake and demand, a widespread monitoring system of the produced and consumed power over time is necessary, as well as efficient forecasting algorithms of datacenters load consumption are required to optimize the usage of energy storage systems (ESS) that collects the surplus of green energy for future needs.

Variability and fast-changing characteristics of applications, for instance, scale-out applications [17] (e.g., web search, MapReduce, etc.), affect the energy consumption of servers due to the dependency on external factors, e.g., number of clients/queries in the system. To this end, the impact of servers' energy consumption on the usage of green energy becomes more substantial, and management of consumed energy will play a major role in lifetime and operation of ESS. Consequently, without consideration of minimizing datacenter energy consumption, many existing approaches to management of green energy and batteries are suboptimal.

In this chapter, we introduce and propose a multilevel and multi-objective framework for the optimization of green virtualized datacenters, to jointly minimize the energy consumption and the carbon footprint, exploiting renewable energy sources, state-of-the-art VMs allocation schemes and Hybrid Electric Systems (HES). With HES, we refer to electrical ESS where different battery technologies are employed together, allowing to compensate for the inherent drawbacks of

each technology. We incorporated dynamic VM allocation into servers' powers by novel HES, and optimization methods to maximize the battery bank lifetime are used. The framework consists of two modules running concurrently: the datacenter energy controller which minimizes the energy consumption of datacenter without any significant quality-of-service (QoS) degradation and shares the real energy consumption data with the green energy controller and the green energy controller that manages renewable sources and HES, providing feedback to the datacenter energy controller.

The datacenter energy controller is based on a state-of-the-art correlation-aware VM allocation scheme [21] due to a high correlation within a cluster of applications in virtualized datacenters. Regarding load correlation, the authors demonstrate that having detailed information about the applications characteristics, as opposed to using stationary load values for the VMs (e.g., peak or average values), gives the opportunity to further reduce the energy consumption of a datacenter. On the other side, the QoS degradation occurs when the aggregated utilization among colocated VMs is beyond the CPU capacity of a server. It means that there will be some workloads which cannot be executed at the right time. Therefore, datacenter providers take into account the service-level agreements requirements to satisfy the customers. The green energy controller, based on [32], is a two-phase controller that takes into account the cost policies of the grid energy and exploits forecasts of both the datacenter's load and of the incoming energy from renewables. The framework uses PV modules as green energy source and two battery technologies (lead-acid and lithium-ion) for the HES that are used with different priorities and roles.

In current datacenters, not enough efforts have been dedicated to implement adaptive energy reduction techniques and real-time resource scheduling to manage efficiently IT equipment and renewable energy sources. The novelty of our work consists in the introduction of a HES architecture to replace standard uninterruptible power supply (UPS) systems, which allows an active management and the full exploitation of the energy buffers for the locally generated renewable energy. We also designed a dedicated control loop which connects the VMs allocation scheme to the HES manager and optimizes the resources in real time. At the same time, the modular structure allows to use both general-purpose models and high-end ones for performance evaluation, model verification, and feasibility analysis.

35.2 Related Work

Renewable energy sources integration in the electricity grid and in particular green datacenters are currently a hot topic. Different research ideas have been presented in the last few years that address the problem of exploiting local energy generation to mitigate grid energy demand of datacenters [18] and in general of any human activity [12]. At the same time, HES have been addressed in several works available

in the literature. The fundamental idea behind HES management is to use batteries as energy buffers to store the amount of green energy that cannot be used directly by the connected loads. Different management approaches have been proposed to automatically control the energy flows from renewables to loads and storage units [16], and also hybrid solutions for battery banks have been demonstrated [39]. This is particularly of interest nowadays because of the large availability of second-life batteries from electric vehicles that can have up to 75% remaining capacity available for storage applications [25, 40]. Despite the market availability of hybrid storage systems is still far, the literature review demonstrates that these technologies are worth the efforts for being implemented. In this work, we followed the approach proposed in [32] to shape the active UPS (or HES) system presented in the following. The authors in [32] propose a two-phase control scheme that exploits intrinsic advantages of different battery technologies mitigating, at the same time, their drawbacks.

A number of research works present methods for server consolidation based on per-VM workload characteristics, i.e., the peak, off-peak, and average utilization of workload [26, 35], which aims to reduce heat dissipation of hot spot zones and improve overall power utilization in datacenters [9, 22]. In [34], authors propose abstract models to balance computing power in a datacenter by minimizing peak inlet temperatures. A holistic approach that manages IT, power, and cooling equipment by dynamically migrating servers' workloads and adjusting cooling is presented in [13]. Experimental results for a virtual datacenter demonstrate a reduction by 35% in power consumption and 15% in cooling. Authors in [27] present a control-oriented model that considers cyber and physical dynamics in datacenters to study the potential impact of coordinating the IT and cooling controls. To achieve further power savings while maintaining the QoS level, joint relationships among VMs, like load correlations, have been exploited in recent works [19, 24, 36]. For instance, in [24], Meng et al. proposed a VM sizing technique that pairs two uncorrelated VMs into a super-VM by predicting the workloads. However, once the super-VMs are formed, this solution does not consider dynamic changes of the VMs' load, which limits further energy savings. Therefore, these approaches do not work well with non-stationary and fast-changing VM behaviors in particular for scale-out applications. In [21], a power-efficient solution is proposed based on the first-fit-decreasing heuristic to separate load-correlated VMs especially targeting the characteristics of the scale-out applications. They also exploit server's dynamic voltage and frequency scaling Dynamic Voltage and Frequency Scaling (DVFS) techniques to achieve further energy savings. Note that these schemes do not take into account the renewable energy sources and datacenter system model in modern green datacenters.

There is no evidence in the literature of the joint application of HES optimization and correlation-aware techniques to the optimization of datacenter energy consumption, and the potential savings (both from environmental and money perspectives) are clearly worth the effort for further investigation.

35.3 The System Modeling Framework

We introduce a novel green datacenter system model where datacenter equipment, PV modules, smart grid, and UPS are connected as shown in Fig. 35.1. The IT equipment and cooling system inside this datacenter are the major contributors to power consumption than the other facilities. These components are combined using Power Distribution Units (PDUs) that eventually connect to the Charge Transfer Interconnect (CTI) bus that serve the whole facility [15]. In this framework, the UPS is designed as a HES to provide both supply in case of grid outages and a buffer for green energy.

The system models two battery banks, a PV module and the bidirectional CTI bus, managed by a dedicated controller, not shown, as presented in [37]. Each unit is connected to the CTI by means of a bidirectional DC-DC converter for level shifting and charge routing, while the PV's one is unidirectional. Grid and PDUs are modeled in terms of power source and load, connected with the CTI by means of AC-DC and DC-AC converters, respectively.

We defined two constraints to the simulated system: (i) the exceeding renewable energy cannot be injected into the main grid (if it cannot be stored) and (ii) a

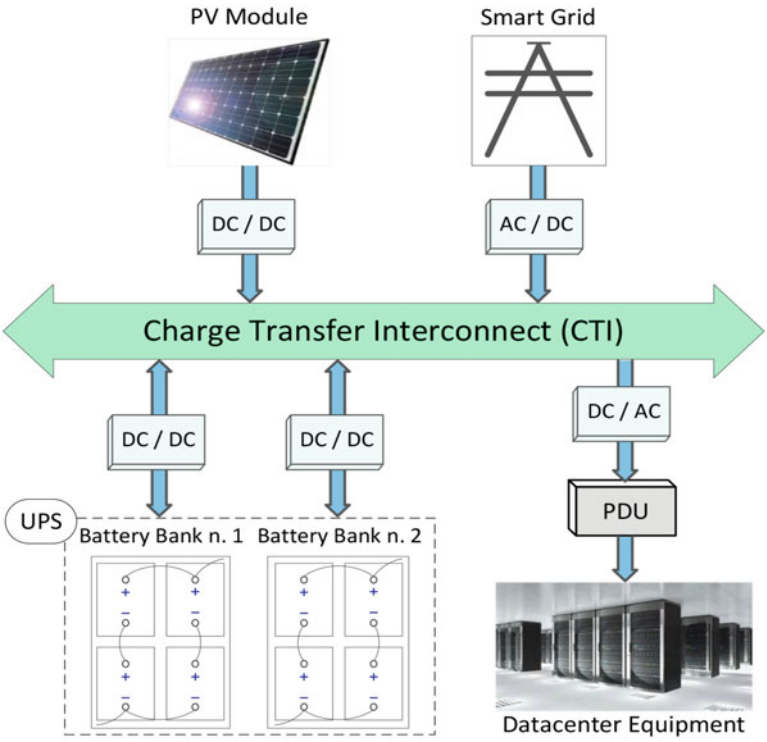


Fig. 35.1 The complete system modeling framework

peak/off-peak price scenario from a regulated electricity market for the energy taken from the grid (we considered the Zurich's tariff 7.5/14.9 CHFcent/kWh [1]).

Thus, renewable energy and batteries should completely sustain the load of the datacenter or, at least, provide supply during outages and periods with the highest price. These choices are justified by the fact that selling energy back to the grid, namely, providing net-metering ancillary service to the Distribution System Operators (DSOs), follows rules that are country-specific and strongly depend on the interface between datacenter and energy network; moreover, datacenters are usually big energy consumers, and it is unlikely to have enough excess green energy to justify the effort (economically and technologically) of improving the electric system to handle this task. The peak/off-peak price scenario in a regulated energy market instead can be easily implemented also in a free energy market scenario where the energy price is continuously evolving; in this case, our assumption can be seen as a threshold on the freely variable price: while the free market price is below the threshold, it is more convenient to buy from the grid, and the opposite when the price rises.

We developed a discrete-time framework (cf. see ▶ Chap. 6, "Optimization Strategies in Design Space Exploration" for more details on different design space exploration options) that simulates the target green datacenter, with hourly time steps. The green energy controller manages the PV modules, the heterogeneous batteries and the CTI, the HES considered in this framework, and has been implemented using MATLAB. The datacenter energy controller, implemented in C++, manages the datacenter and VMs allocation scheme. Both components communicate using sockets for interprocess communication, while the time-step length of one hour guarantees that the time for VMs relocation (several GBs) does not overtake the actual execution time.

35.3.1 Energy Management Models

According to Fig. 35.1, the power management problem is solved at the CTI bus level which is a DC path. Conversely, the system comprises both AC and DC sources/loads; thus, for the former ones, it is required to consider the power factor component in the conversion. For example, considering the power intake from the grid, if we measure the total apparent power that enters the rectifier, for example, on the grid side $P_{\text{Grid}}[VA] = V_{\text{RMS}} \cdot I_{\text{RMS}}$, this can be converted into active power (the useful power available on the DC side) according to the $P_{\text{Grid}}[W] = P_{\text{Grid}}[VA] \cdot \cos(\phi)$ where ϕ is the angle between voltage and current waveforms and $\cos(\phi)$ is called power factor. In addition, the converter's efficiency $\eta_X(.)$ must be added to any transformation, since it depends on the actual power flowing with respect to the nominal one.

$$P^{\text{CTI}}_{\text{Datacenter}}(t) = P^{\text{CTI}}_{\text{Grid}}(t) + P^{\text{CTI}}_{PV}(t) + \sum_{n=1}^{\text{nEES}} \alpha \cdot P^{\text{CTI}}_{\text{EES},n}(t) \tag{35.1}$$

$$P_{\text{Grid}}^{\text{CTI}}(t) = P_{\text{Grid}}(t) \cdot \cos(\phi) \cdot \eta_{\text{ACDC}}\left(\rho(t)\right) \tag{35.2}$$

$$P_{PV}^{\text{CTI}}(t) = P_{PV}(t) \cdot \eta_{\text{DCDC}}\left(\rho(t)\right) \tag{35.3}$$

$$P_{\text{EES},n}^{\text{CTI}}(t) = P_{\text{EES},n}(t) \cdot \eta_{\text{DCDC}}\left(\rho(t)\right) \tag{35.4}$$

$$P_{\text{Datacenter}}^{\text{CTI}}(t) \cdot \eta_{\text{DCAC}}\left(\rho(t)\right) = P_{\text{Datacenter}}(t) \tag{35.5}$$

$$\rho(t) = \frac{P_{\text{out}}}{P_{\text{nom}}} \cdot 100 \tag{35.6}$$

Equation (35.1) represents the power balance of the system, it states that the sum of the input from the grid, PV and battery arrays must be equal to the datacenter requirements, additionally the α is a directional parameter which can be $-/+1$ depending on the charging/discharging status (source or load of the system), and nEES is the number of separated battery banks that compose the HES. Equations (35.2), (35.3), (35.4) and (35.5) describe the AC-to-DC and DC-to-DC conversion functions used for each system component, where the conversion efficiency term $\eta_X(.)$ depends on $\rho(t)$, the ratio of power requested by the system with respect to the nominal power delivered by the converter, which is expressed in percentage as defined by Eq. (35.6).

In order to reduce the computational complexity and generalize the system's models, we considered fixed power factor equal to one; fixed CTI voltage level and energy converters have been modeled considering a fixed 90% efficiency since detailed efficiency curves for high-power equipment are not publicly provided by manufacturers [2] but still are claimed to work in the range of 80–95% (with loads down to $\rho(t) = 20\%$).

35.3.2 Electrical Energy Storage System

The HES can exploit two heterogeneous battery banks managed in hierarchical fashion: a lead-acid array (the battery bank n. 1) and a lithium-ion array (the battery bank n. 2). The battery model is based on the Peukert's law [29]. The goal is to model HES that combine the advantages of the different battery technologies (lead-acid and lithium-ion). The module, as all the modules in the framework, has been conceived as a plug-and-play component; therefore, it can be easily replaced and adapted.

Equation (35.7) defines the State of Health (SoH) of the battery as a ratio between currently available charge capacity (C_{ref}) and the nominal one. Equation (35.8) defines the charge capacity as a linear combination of the previous charge and a term that depends on the charge which is drained, where C_{nom} is the nominal charge declared by manufacturer while Z_b, linear aging coefficient, is a parameter depending on the battery technology [30]. The following two equations (Eqs. 35.9 and 35.10) allow to determine the State of Charge (SoC) and the equivalent battery current (I_{eq}), function of the current flowing from batteries (I), with respect to the

nominal battery parameters: I_{ref} the reference discharge current (provided by the manufacturer and used to compute the reference charge), the Peukert's coefficient k_b and the charge actually used by the system, computed as current I_{eq} times time slot (t_{slot}) length in seconds. The SoH of the battery decreases only during discharge, so it is calculated only during discharge, whereas the SoC is updated during both charge and discharge cycles. More details about the model and its utilization can be found in [29, 30].

$$\text{SoH}(t+1) = \frac{C_{\text{ref}}(t+1)}{C_{\text{nom}}} \tag{35.7}$$

$$C_{\text{ref}}(t+1) = C_{\text{ref}}(t) - C_{\text{nom}} \cdot Z_b \cdot (\text{SoC}(t) - \text{SoC}(t+1)) \tag{35.8}$$

$$\text{SoC}(t+1) = \frac{C_{\text{ref}}(t) \cdot \text{SoC}(t) - (I_{\text{eq}}(t) \cdot t_{\text{slot}})}{C_{\text{ref}}(t)} \tag{35.9}$$

$$I_{\text{eq}}(t) = \left(\frac{|I(t)|}{I_{\text{ref}}}\right)^{(k_b-1)} \cdot I(t) \tag{35.10}$$

We tuned the parameters of the general-purpose model (maximum and reference charge/discharge currents) according to commercial devices, a VARTA Professional Dual Power (230 Ah @ 12 V) [3] as the lead-acid, and a StarkPower "UltraEnergy" (100 Ah @ 12 V) [4] as the lithium-ion.

We preferred to double the size of the battery bank n. 1, with respect the lithium-ion one, because lead-acid technology is cheaper, easier to recycle, and has a wider working temperature range. However, lead-acid batteries suffer from a limited number of sustainable cycles (i.e., lifetime). The lithium-ion technology instead offers at least one order of magnitude higher number of cycles, but it is also more expensive. To maximize the lifetime of the storage (in particular of the lead-acid bank), we put some constraints on the allowed Depth-of-Discharge (DoD) for both banks. To force both banks to work in the optimal range of SoC, we set the minimum SoC to 65% for the bank n. 1 and 70% for the bank n. 2. The remaining capacity is however available in the event of outage, thus providing standard UPS support.

Moreover, in the simulations, we considered two configurations, the HES-1 where we have 48 kWh as lead-acid capacity (16.8 kWh available) and 24 kWh as lithium-ion capacity (7.2 kWh available) and the HES-2 with 96 kWh (33.6 kWh) and 48 KWh capacity (14.4 kWh), respectively.

35.3.3 Photovoltaic Module

The PV module provides green energy accordingly to the intensity of the solar irradiance impinging on it, which in turn depends on the weather mostly. In this framework, we implemented it as a linearly varying voltage source, with integrated MPPT controller [32] and tuned accordingly to real device's characteristics [5]. Sun

irradiance [6] and temperature profiles [7] for the year 2005 in Zurich have been used for tests.

$$P_{PV} = \left[P_{\text{PV,STC}} \cdot \left(\frac{G_T}{1000} \right) \cdot \left(1 - \gamma \cdot (T_j - 25) \right) \right] \cdot N_{\text{PV},S} \cdot N_{\text{PV},P} \tag{35.11}$$

$$T_j = T_{\text{amb}} + \left(\frac{G_T}{800} \right) \cdot \text{NOCT} - 20 \tag{35.12}$$

Equation (35.11) presents the linear model of the PV array, the parameters were evaluated in Nominal Operating Cell Temperature (NOCT) and Standard Test Conditions (STC) which are the nominal output power ($P_{\text{PV,STC}} = 2.65\,\text{W}$) in this case, the cell temperature (T_j), irradiance level ($G_T = 1000\,\text{W/m}^2$ @ 25 °C), and the temperature coefficient ($\gamma = 0.0043\%/°\text{C}$), while $N_{\text{PV},S}$ and $N_{\text{PV},P}$ are the number of series and parallel cells in the module. The cell temperature is then obtained using Eq. (35.12), where T_{amb} is the environmental temperature, $G_T = 800\,\text{W/m}^2$ @ 20 °C and NOCT = 45.5 °C.

We tuned the PV module size considering two different cases of peak power production (hence the number of cells and panels) that are 10 kWp for the HES-1 simulating scenario and 30 kWp for the HES-2.

35.4 Simulation Framework Description

The overall diagram of our simulating framework that jointly manages the green energy and datacenter Energy Controllers is shown in Fig. 35.2. At the beginning of the simulation time horizon (off-line phase), the green energy controller computes the expected energy budget for the datacenter, processing historical datacenter power profiles as well as the sun irradiance forecasts. This task is executed only once and provides a preliminary energy budget for the whole simulation horizon.

The online phase starts when the off-line phase of the green energy controller sends the available energy budget to the datacenter energy controller for the first time slot. Next, it waits until the VMs allocation is completed according to the prediction of upcoming loads of VMs and then receives back the real energy demand of the datacenter computed based on the real workload. Therefore, the green energy controller compensates the differences between (i) expected and available green energy and (ii) real energy consumption and energy budget for the datacenter, using the lithium-ion battery as additional energy reserve or the grid if both banks in the HES have been drained. To this end, if the actual energy consumed by datacenter is higher than the expected, the green energy controller compensates the datacenter energy requirements. At the end of each time slot, the green controller provides an updated budget to the datacenter energy controller for the VM allocation of the next time slot.

On the other side, the datacenter energy controller tries to find the best allocation for VMs on the servers at each time slot using the VMs specification from the

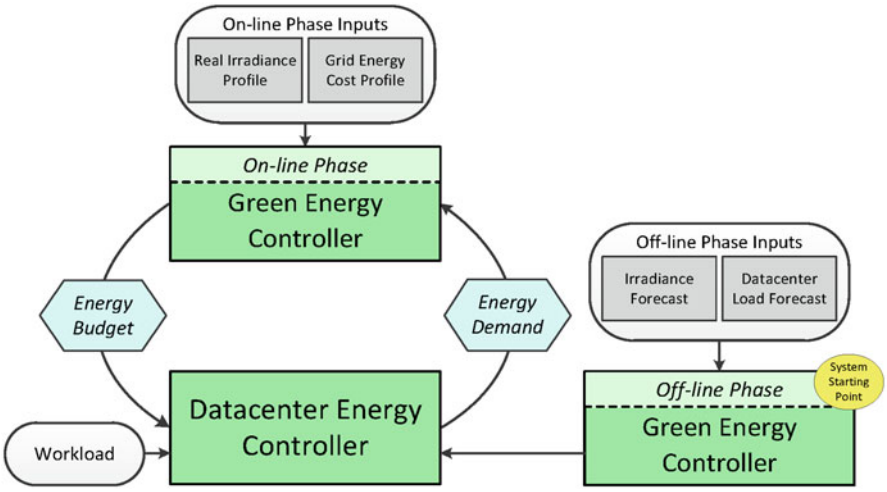


Fig. 35.2 The simulating framework that jointly manages the green energy and datacenter energy controllers. Off-line phase, as a starting point of simulation, is executed once at the beginning of the simulation time to compute expected energy budget for datacenter. In online phase, at each time slot, datacenter energy controller first receives forecasted workload and energy budget from green energy controller to allocate VMs to servers and then sends back the real energy demand to green energy controller

previous time slot as incoming workload and the energy budget provided by green energy controller. The goal is to allocate VMs to the minimal number of servers that yields in optimized total energy consumption of datacenter, as it will be explained in the following. After the allocation was completed, the datacenter energy controller communicates the actual energy demand for the current time slot to the green energy controller. Both of the controllers are invoked periodically, at every time slot, i.e., t_{slot}. The overall process of the framework and two controllers' communication have been shown in Fig. 35.3. In the following sections, we describe these two controllers in detail.

35.4.1 Datacenter Energy Controller

In this section, we have considered the state-of-the-art correlation-aware VM allocation scheme as a datacenter power management solution [21]. Correlation refers to the VMs' utilizations when the peaks of two VMs occur at the same time during a certain time interval. Therefore, for using the server's resources efficiently during a time slot, highly correlated VMs should be placed apart, in different servers. Thereby, based on the VMs' utilization patterns, the aggregated utilization of colocated VMs nearly reaches their server's capacity during a time slot. This favors consolidation and leads to power savings by lowering the number of active servers. In this context, due to the distributed operations of multiple VMs

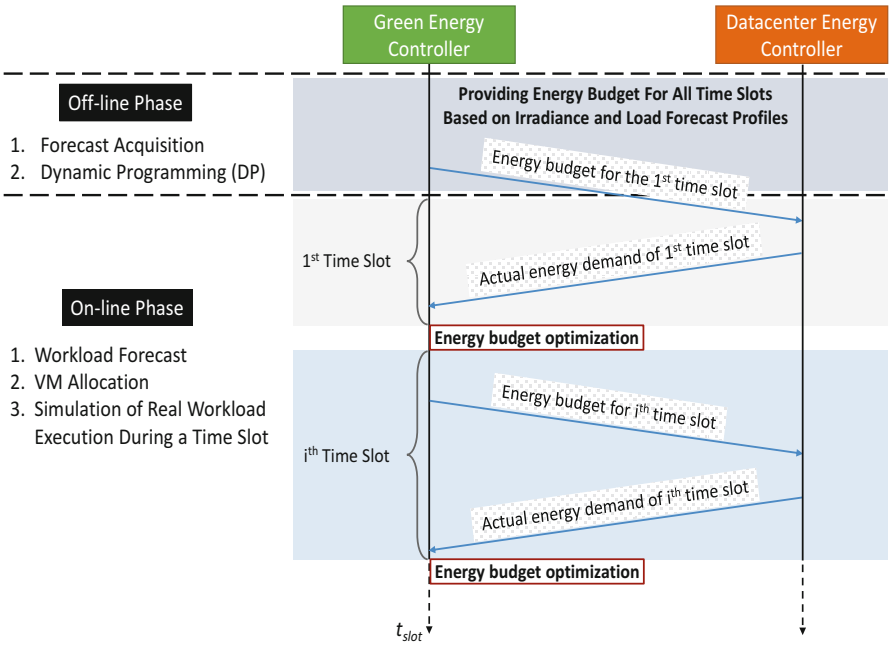


Fig. 35.3 Overall process of the proposed framework – joint datacenter and Green Energy Controllers

in a cluster, a high correlation within a cluster of VMs is observed, called intra-cluster correlation, rather than the correlation among different clusters targeted in other correlation-aware schemes [19, 36]. The correlation-aware VM allocation method has been proposed, in [21], while sharing cores among colocated VMs based on defining a cost function depending on QoS requirement to efficiently quantify the correlation between the VMs across a certain time horizon. Finally, a way to scale the Voltage/Frequency (V/f) level is provided to achieve more power savings without any QoS degradation. In this algorithm, the VMs are allocated such that the correlation among the allocated VMs in the server is minimized, while the server does not exceed its total CPU capability, as well as the number of the active servers is minimized while satisfying performance requirements. Once all the VMs are allocated into servers, an optimal V/f level for each server is determined. This correlation-aware VM allocation algorithm is periodically invoked at every t_{slot}.

35.4.2 Green Energy Controller

The green energy controller is a two-phase scheduler – off-line and online phases – that manages the CTI bus and provides guidelines to the datacenter energy con-

troller, by recursively solving the set of equations presented in Sect. 35.3. Moreover, see ▸ Chap. 10, "Design Space Exploration and Run-Time Adaptation for Multicore Resource Management Under Performance and Power Constraints" for more details about combined design- and run-time exploration and adaptation approaches for computing systems.

The off-line phase's goal is to find the best resource allocation strategy to minimize the energy intake from the grid (i) and to maximize the lifetime of the lead-acid battery bank (ii) by minimizing the number of charge-discharge cycles and using as much as possible uninterrupted cycles. This is based on Dynamic Programming (DP) that is a strategy to solve complex problems by splitting them into lower complexity ones, solving and storing each solution; thus, when a previously solved problem occurs, the system looks up the previous solution saving computational time. It takes as input the expected workload of the datacenter, the price profile of the energy from the grid, and the irradiance forecasts for the whole time horizon [32, 38]; in this phase, the scheduler manages the battery bank n. 1 only. The algorithm ranks all the possible system states (charge to discharge, charge to charge, discharge to charge, and discharge to discharge) for each time slot in the simulation horizon that fulfills the above constraints. For each state transition, it assigns a weight based on the battery usage; the higher the weight the lower the ranking. At the end, it provides an optimal energy budget for each time slot and the best utilization strategy for the lead-acid bank for the whole time horizon. Only the budget for the first time slot is then sent to the datacenter energy controller and this message triggers the online phase. All the other energy budgets computed are kept in memory for the online phase to use them when the off-line concludes.

The on-line phase, for each time slot, optimizes the initial energy budget, computed by the off-line phase, trying to compensate the difference between expected workload and irradiance forecast with respect to the real data measured by the system. In the online phase, the scheduler manages also the battery bank n. 2 mainly to compensate error in the forecasts and to maximize the lifetime of the lead-acid bank. This is a constrained multivariate optimization problem that has been solved numerically using the Matlab's fmincon [8] solver. For each time slot, the green scheduler must find the optimal current balance in the CTI to minimize the energy taken from the grid (optimization goal), to fulfill the off-line lead-acid battery scheduling and to supply the load. For each component of the system (grid, PV, batteries and load), we set constrained boundaries for the currents and the input power from the grid, linear constraints for the CTI based on the Kirchhoff currents law, and nonlinear constraints to compute the effect of energy converters and batteries' SoC. Problem's constraints (current flow direction for batteries and use of the grid) change in accordance with the system state, in this way it is possible, for example, to force the lithium-ion battery to be discharged when the lead-acid battery is recharged and the green energy is unavailable or lower than the load. At the end of the time slot, the actual energy balance is updated to the datacenter, and this triggers also a new cycle of the simulator with the following t_{slot}.

35.5 Experimental Results

We validated the effectiveness and applicability of the proposed framework to larger-scale problems using 2-week simulation horizon, workload traces obtained from a real datacenter setup, and real irradiance and temperature profiles. We arranged the simulations in two separate sets: firstly we evaluated the best VM allocation algorithm in terms of energy and QoS; secondly we placed this best scheme into the datacenter energy controller, and we executed the joint optimization framework.

35.5.1 Setup

We modeled a green urban datacenter consisting of medium-sized facilities with two components: computing power consumption (IT equipment) (i) and Computer Room Air Conditioning (CRAC) power consumption as the cooling unit (ii). We evaluated the effectiveness of the proposed solution with a virtual testbed consisting of 250 servers where the servers are homogeneous. We targeted an Intel Xeon E5410 server configuration which consists of eight cores and two frequency levels (2.0 and 2.3 GHz) and used the power model proposed in [28].

To simulate the datacenter workload and energy demand, we sampled the CPU utilization of a real datacenter setup every 5 min for 1 day; then we duplicated the samples up to 14 days. Such assumption has been proved by real-trace studies, since the real datacenter's workload shows significant variability and a daily pattern during 1 week [23]. Finally, to generate different samples for each day, we synthesized fine-grained samples per 5 sec with a lognormal random number generator [10], whose mean is the same as the collected value for the corresponding 5 min sample rate.

We computed the irradiance forecasts implementing the algorithm presented in [11]; an example of the two resulting sequences is depicted in Fig. 35.4. At the

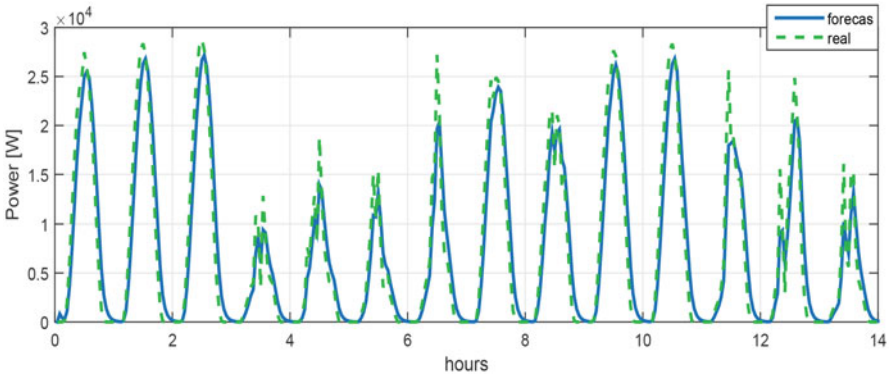


Fig. 35.4 Solar power profile, forecasted vs. real

same time, we used hourly averaged energy consumption profile from the real datacenter as forecast, which results in a smoothed profile compared to the original one.

35.5.2 Results

As previously introduced, we split the performance evaluation in two separate sets of experiments. To select the best VM allocation scheme for power management to use with the datacenter energy controller, we compared the following three approaches:

- Best-Fit-Decreasing (BFD): a conventional best-fit-decreasing heuristic approach. In detail, after sorting VMs in decreasing order of their utilization, the algorithm allocates each VM to a server that provides the closest resource requirements with respect to this VM utilization (i.e., the server with the smallest remaining capacity is sufficient to contain the VM).
- Peak Clustering-based Placement (PCP) [36]: a correlation-aware VM allocation which clusters VMs using its envelope-based correlation classification. The authors presented a static clustering-based VM allocation method by defining VM utilization in a time series as a binary sequence where the value becomes "1" when utilization is higher than a threshold value, otherwise "0". This algorithm first clusters VMs such that the envelopes of VM utilization included in different clusters do not overlap. Then, it allocates VMs to servers in order to colocate VMs in different clusters.
- Correlation-aware VM Placement (CVMP) [21] the correlation-aware VM allocation considered as the state-of-the-art approach and explained in Sect. 35.4.1.

Figure 35.5 compares the total energy consumption of the three approaches under different number of VMs (obtained by duplicating the trace for 250 VMs) in the system for a horizon of 14 days when we set the V/f level at the time of VM placement t_{slot}. The CVMP algorithm provides up to 11.6 and 7.3% energy savings compared to BFD and PCP, respectively, due to using the lower frequency levels more frequently. It is noteworthy that PCP provides almost similar results with BFD because, due to high and fast-changing correlations among VMs in our utilization traces, PCP classifies VMs into only one cluster during most of the time periods. When the number of clusters is one, PCP behaves exactly the same as BFD. Note that the semi-linear trend of the energy consumption depends on the analogous behavior of the workload among different days, in a typical datacenter.

Table 35.1 shows the maximum violation defined as maximum per-period ratio of the number of over-utilized time instances (i.e., when the aggregated utilization among colocated VMs is beyond the CPU capacity of a corresponding server) to t_{slot}, during the two weeks under different number of VMs in the system. A graphical representation of these data is provided in Fig. 35.6. As a result, the CVMP scheme provides a drastic reduction of the violations, up to 10.4 and 9.6%

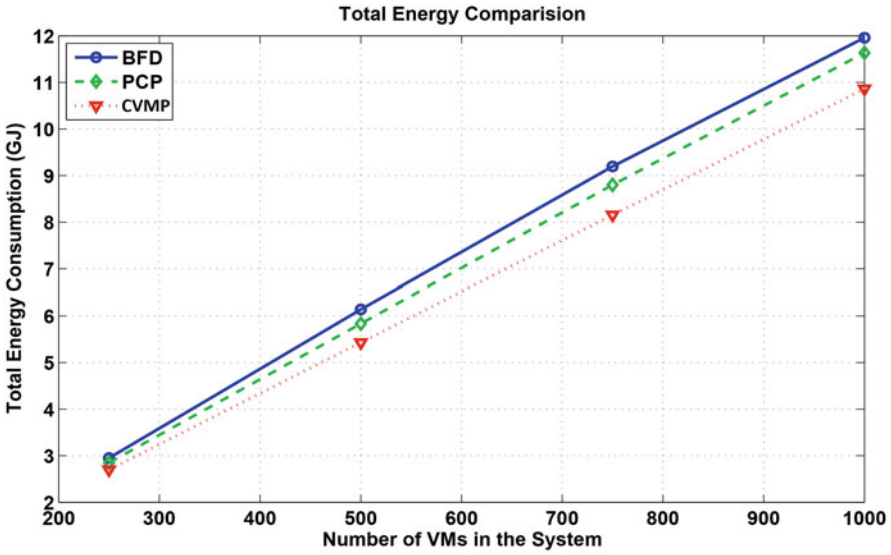


Fig. 35.5 Total energy consumption of datacenter under different number of VMs for a horizon of 14 days

Table 35.1 Maximum violations (%) of ratio of over-utilized time instances to t_{slot}, during the entire periods, i.e., 336 h (14 days) under different number of VMs scenario

Approach	Number of VMs			
	250	500	750	1000
BFD	2.1	4.9	9.6	18.4
PCP	1.1	2.8	3.4	17.6
CVMP	0.85	2	3.1	8

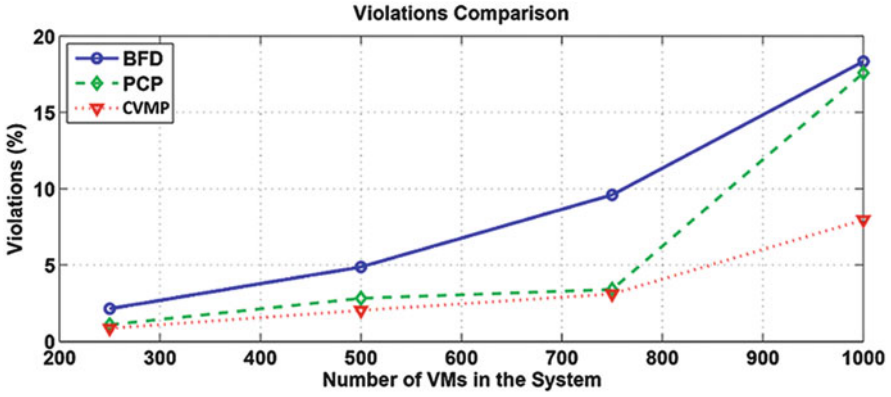


Fig. 35.6 Trend of maximum violations (%) under different number of VMs for a horizon of 14 days

Table 35.2 Overall framework results in terms of economic benefit of renewable-enabled datacenter with respect to a grid-connected one. Two HES configurations are evaluated, HES-1 with 48 kWh as lead-acid and 24 kWh as lithium-ion capacity and HES-2 with 96 kWh and 48 kWh capacity, respectively

Configuration	Winter savings (PV only)	Summer savings (PV only)
250 VMs		
HES-1	29.30% (25.54%)	76.46% (57.86%)
HES-2	62.22% (38.72%)	96.13% (66.45%)
500 VMs		
HES-1	14.30% (13.16%)	55.92% (48.00%)
HES-2	38.43% (31.30%)	85.28% (61.59%)
750 VMs		
HES-1	9.53% (8.76%)	43.49% (40.16%)
HES-2	27.69% (24.86%)	73.39% (57.35%)
1000 VMs		
HES-1	7.05% (6.57%)	33.34% (32.51%)
HES-2	20.64% (19.16%)	65.28% (53.96%)

compared to BFD and PCP, respectively. In CVMP method, VMs are allocated based on their peak utilizations, which were predicted from their history. Despite the provision based on the peak utilization, we observed quality degradation over the three approaches due to the mis-predictions of the peak utilization, especially during abrupt workload changes under increasing the number of VMs in the system. However, the CVMP method can statistically reduce the probability of the violation by colocating uncorrelated VMs. Thus, the probability of joint under-predictions among the colocated VMs is drastically decreased. Using the CVMP algorithm, we performed the complete framework simulation (VM allocation, green energy scheduling, and communication between the two controllers) with $t_{\text{slot}} = 1$ h, with predictions of upcoming workloads of datacenter using a last-value predictor.

Table 35.2 summarizes the results in terms of cost savings depending on the number of VMs, the HES size, and the season. The cost savings are computed as the difference between electricity cost to sustain the datacenter workload with or without the renewable energy sources. As expected with larger battery capacities (HES-2 configuration), we get higher savings. We compared also with the cost saving of using the PV panels without any storage (between brackets) to demonstrate the advantage of the proposed approach. Although in winter scenario the low irradiance and the cold weather strongly impact the renewable energy generation, causing the batteries to rarely reach the full charge, they still provide advantages in terms of savings. During summer instead the batteries are fully exploited resulting in higher savings with respect to the previous scenario. According to the model, during summer, when the HES system's usage is more intensive, we experienced a maximum SoH decrease of 0.07% (ratio between nominal and remaining capacity), which means a lifetime longer than 15 years to reach the 70% of nominal capacity

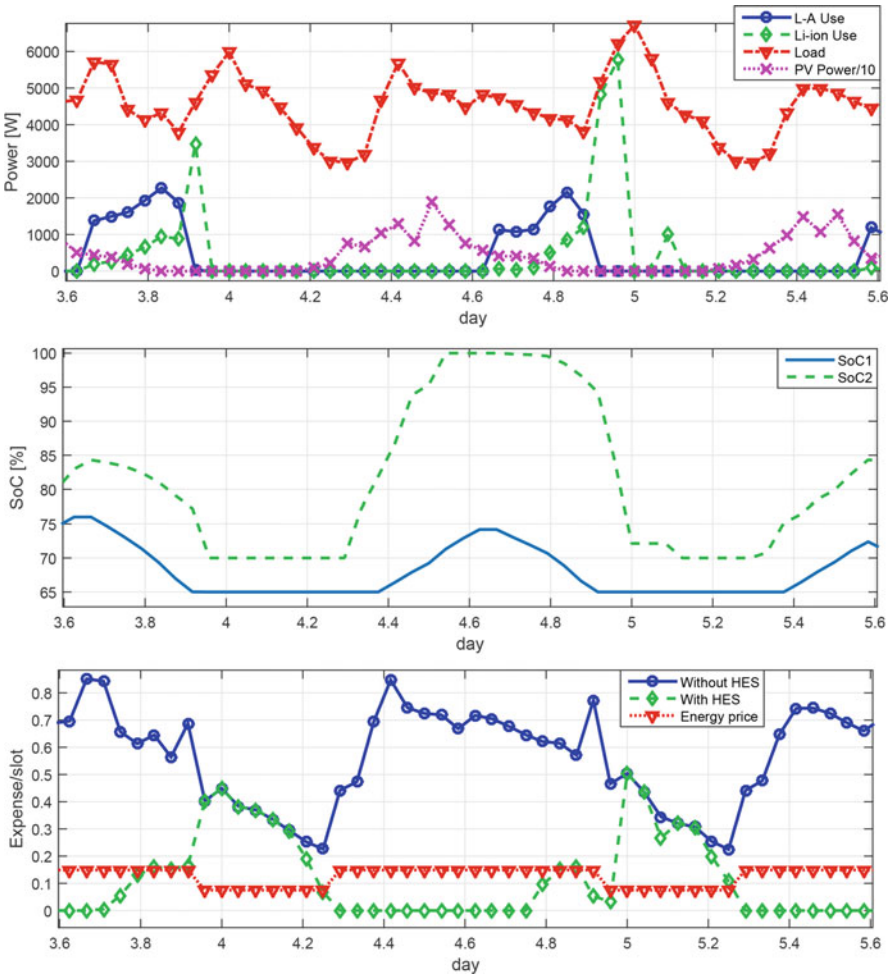


Fig. 35.7 Two-day framework evolution with 500 VMs, HES-2 (96 kWh lead-acid and 48 kWh lithium-ion capacity) configuration, and summer irradiance (48 time slots). Power profile of the datacenter components (*top*); percentage SoC of the battery bank n. 1 (SoC1) and n. 2 (SoC2) (*middle*); cost per time slot (*bottom*)

(lead-acid battery near the end of life). Finally, Fig. 35.7 shows a 2-day view (48 time slots) of the framework evolution with 500 VMs, summer irradiance, and HES-2 configuration. We can observe the role of the energy buffer that allows to use green energy when there is no input from the PV panels (Fig. 35.7-top) and the resulting money saving (Fig. 35.7-bottom). In the specific time horizon depicted (Fig. 35.7-middle), we experienced a low level of irradiance compared to other days in the overall horizon (cfr. Fig. 35.4); it results in a lower amount of energy available to recharge the batteries, in particular the battery bank n. 1 which has a bigger

capacity and a smaller recharge current with respect to the lithium-ion one. Similar considerations can be made for the other three cases that are not reported for the sake of summary.

35.6 Conclusion

In this chapter, we have presented a novel dynamic and multi-objective framework to manage the energy consumption of datacenter, battery banks lifetime, and energy bill cost. The datacenter energy controller minimizes the total energy consumption using the state-of-the-art correlation-aware VM allocation scheme for the given VMs' specifications and energy budget provided by the green energy controller while improving QoS requirements. In the green energy controller, we use a real-time optimization technique to maximize the lifetime of battery banks and to reduce the energy bill by managing the PV source, in price-varying scenarios, and considering the energy consumed by the datacenter. Finally, we validated the effectiveness and applicability of our proposed system with the utilization traces obtained from a real datacenter setups. Our experimental results show that the proposed framework provides up to 11.6% energy savings and up to 10.4% improvement of QoS level compared to existing conventional solutions under different number of VMs in the system and up to 96% money saving in the electricity bill.

Acknowledgments This work has been partially supported by the EC FP7 GreenDataNet STREP Project (Agreement No. 609000) and the YINS RTD Project (no. 20NA21_150939), funded by Nano-Tera.ch with Swiss Confederation Financing and scientifically evaluated by SNSF.

References

1. [online] http://www.strompreis.elcom.admin.ch/PriceDetail.aspx?placeNumber=261&OpID=565&-Period=2014&CatID=12
2. [online] http://www.schaeferpower.de/cms/en/produkte.html
3. [online] http://www.varta-automotive.com/en-gb/products/industrial/industrial-professional-dual-purpose
4. [online] http://www.starkpower.com/spnews/energystoragebatt
5. [online] http://www.enfsolar.com/pv/cell-datasheet/429
6. [online] http://www.soda-is.com/eng/services/services_radiation_free_eng.php
7. [online] http://www.tutiempo.net/en/Climate
8. [online] http://www.mathworks.com/help/optim/ug/fmincon.html
9. Bash C, Forman G (2007) Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center. In: 2007 USENIX annual technical conference on proceedings, ATC'07. USENIX Association, Berkeley, pp 29:1–29:6
10. Benson T et al (2010) Understanding data center traffic characteristics. ACM SIGCOMM Comp Commun Rev 40(1):92–99
11. Bergonzini C et al (2010) Comparison of energy intake prediction algorithms for systems powered by photovoltaic harvesters. Microelectron J 41(11):766–777

12. Carpinelli G, Celli G, Mocci S, Mottola F, Pilo F, Proto D (2013) Optimal integration of distributed energy storage devices in smart grids. IEEE Trans Smart Grid 4(2):985–995
13. Chen Y, Gmach D, Hyser C, Wang Z, Bash C, Hoover C, Singhal S (2010) Integrated management of application performance, power and cooling in data centers. In: Network operations and management symposium (NOMS). IEEE, pp 615–622
14. Clark J (2013) It now 10 percent of world's electricity consumption, report finds. [online] http://www.theregister.co.uk/2013/08/16/it_electricity_use_worse_than_you_thought/
15. Deng N et al (2011) Concentrating renewable energy in grid-tied datacenters. In: 2011 IEEE international symposium on sustainable systems and technology (ISSST). IEEE, pp 1–6
16. Farhangi H (2010) The path of the smart grid. Power Energy Mag IEEE 8(1):18–28
17. Ferdman M et al (2012) Clearing the clouds: a study of emerging scale-out workloads on modern hardware. ACM SIGARCH Comp Archit News 40(1):37–48
18. Goiri I, Katsak W, Le K, Nguyen TD, Bianchini R (2014) Designing and managing datacenters powered by renewable energy. IEEE Micro 34(3):8–16
19. Halder K et al (2012) Risk aware provisioning and resource aggregation based consolidation of virtual machines. In: 2012 IEEE 5th international conference on cloud computing (CLOUD), pp 598–605
20. Katz RH (2009) Tech titans building boom. IEEE Spectr 46:40–54
21. Kim J et al (2013) Correlation-aware virtual machine allocation for energy-efficient datacenters. In: Design, automation & test in Europe (DATE) conference, pp 1345–1350
22. Leverich J, Monchiero M, Talwar V, Ranganathan P, Kozyrakis C (2009) Power management of datacenter workloads using per-core power gating. Comput Archit Lett 8(2):48–51
23. Liu Z, Chen Y, Bash C, Wierman A, Gmach D, Wang Z, Marwah M, Hyser C (2012) Renewable and cooling aware workload management for sustainable data centers. In: Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on measurement and modeling of computer systems, SIGMETRICS '12. ACM, pp 175–186
24. Meng X et al (2010) Efficient resource provisioning in compute clouds via VM multiplexing. In: Proceedings of the 7th international conference on autonomic computing. ACM, pp 11–20
25. Mukherjee N, Strickland D (2016) Control of cascaded DC-DC converter-based hybrid battery energy storage systems – Part I: stability issue. IEEE Trans Ind Electron 63(4):2340–2349
26. Pakbaznia E et al (2009) Minimizing data center cooling and server power costs. In: Proceedings of the 14th ACM/IEEE international symposium on low power electronics and design. ACM, pp 145–150
27. Parolini L, Sinopoli B, Krogh B, Wang Z (2012) A cyber-physical systems approach to data center modeling and control for energy efficiency. Proc IEEE 100(1):254–268
28. Pedram M et al (2010) Power and performance modeling in a virtualized server system. In: 2010 39th international conference on parallel processing workshops (ICPPW), pp 520–526
29. Riffonneau Y et al (2008) System modelling and energy management for grid connected PV systems with storage. In: 23rd European photovoltaic solar energy conference and exhibition, pp 3447–3451
30. Riffonneau Y, Bacha S, Barruel F, Ploix S (2011) Optimal power flow management for grid connected PV systems with batteries. IEEE Trans Sustain Energy 2(3):309–320
31. Rossi M, Brunelli D (2013) Electricity demand forecasting of single residential units. In: 2013 IEEE workshop on environmental energy and structural monitoring systems (EESMS). IEEE, pp 1–6
32. Rossi M et al (2014) Real-time optimization of the battery banks lifetime in hybrid residential electrical systems. In: Design, automation & test in Europe (DATE) conference, pp 139–145
33. Stewart C et al (2009) Some joules are more precious than others: managing renewable energy in the datacenter. In: Proceedings of the workshop on power aware computing and systems
34. Tang Q, Gupta S, Varsamopoulos G (2008) Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: a cyber-physical approach. IEEE Trans Parallel Distrib Syst 19(11):1458–1472
35. Verma A et al (2008) pMapper: power and migration cost aware application placement in virtualized systems. In: Middleware 2008. Springer, Berlin, pp 243–264

36. Verma A et al (2009) Server workload analysis for power minimization using consolidation. In: Proceedings of the 2009 conference on USENIX annual technical conference. USENIX Association, pp 28–28
37. Wang Y et al (2011) Charge migration efficiency optimization in hybrid electrical energy storage (HEES) systems. In: 2011 international symposium on low power electronics and design (ISLPED), pp 103–108
38. Wang Y et al (2013) Optimal control of a grid-connected hybrid electrical energy storage system for homes. In: Design, automation & test in Europe conference & exhibition (DATE). IEEE, pp 881–886
39. Wang Y, Lin X, Kim Y, Xie Q, Pedram M, Chang N (2014) Single-source, single-destination charge migration in hybrid electrical energy storage systems. IEEE Trans Very Large Scale Integr VLSI Syst 22(12):2752–2765
40. Wang LY, Wang C, Yin G, Lin F, Polis MP, Zhang C, Jiang J (2016) Balanced control strategies for interconnected heterogeneous battery systems. IEEE Trans Sustain Energy 7(1):189–199
41. Zhang Y et al (2011) Greenware: greening cloud-scale data centers to maximize the use of renewable energy. In: Middleware 2011. Springer, pp 143–164

The DSPCAD Framework for Modeling and Synthesis of Signal Processing Systems

36

Shuoxin Lin, Yanzhou Liu, Kyunghun Lee, Lin Li, William Plishker, and Shuvra S. Bhattacharyya

Abstract

With domain-specific models of computation and widely-used hardware acceleration techniques, Hardware/Software Codesign (HSCD) has the potential of being as agile as traditional software design, while approaching the performance of custom hardware. However, due to increasing use of system heterogeneity, multi-core processors, and hardware accelerators, along with traditional software development challenges, codesign processes for complex systems are often slow and error prone. The purpose of this chapter is to discuss a Computer-Aided Design (CAD) framework, called the DSPCAD Framework, that addresses some of these key development issues for the broad domain of Digital Signal Processing (DSP) systems. The emphasis in the DSPCAD Framework on supporting cross-platform, domain-specific approaches enables designers to rapidly arrive at initial implementations for early feedback, and then systematically refine them towards functionally correct and efficient solutions. The DSPCAD Framework is centered on three complementary tools – the Data-flow Interchange Format (DIF), LIghtweight Data-flow Environment (LIDE) and DSPCAD Integrative Command Line Environment (DICE), which support flexible design experimentation and orthogonalization across three major dimensions in model-based DSP system design – abstract data-flow models, actor implementation languages, and integration with platform-specific design tools. We demonstrate the utility

S. Lin (✉) • Y. Liu • K. Lee • L. Li • W. Plishker
Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA
e-mail: slin07@umd.edu; yzliu@umd.edu; leekh3@umd.edu; lli12311@umd.edu; plishker@umd.edu

S.S. Bhattacharyya
Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD, USA

Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland
e-mail: ssb@umd.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_36

of the DSPCAD Framework through a case study involving the mapping of synchronous data-flow graphs onto hybrid CPU-GPU platforms.

Acronyms

ADT	Abstract Data Type
API	Application Programming Interface
BDF	Boolean Data Flow
BPSK	Binary PSK
CAD	Computer-Aided Design
CAL	Cal Actor Language
CFDF	Core Functional Data Flow
CPU	Central Processing Unit
CSDF	Cyclo-Static Data Flow
CUDA	Compute Unified Device Architecture
D2H	Device-to-Host
DICE	DSPCAD Integrative Command Line Environment
DIF	Data-flow Interchange Format
DSP	Digital Signal Processing
FCFS	First-Come First-Serve
FIFO	First-In First-Out
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GLV	Graph-Level Vectorization
GPU	Graphics Processing Unit
H2D	Host-to-Device
HDL	Hardware Description Language
HSCD	Hardware/Software Codesign
ITS	Individual Test Subdirectory
LIDE	LIghtweight Data-flow Environment
MDSDF	Multi-Dimensional Synchronous Data Flow
MILP	Mixed Integer Linear Programming
PREESM	Parallel and Real-time Embedded Executives Scheduling Method
PSDF	Parameterized Synchronous Data Flow
PSK	Phase Shift Keying
PSM	Parameterized Sets of Modes
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature PSK
RVC	Reconfigurable Video Coding
SADF	Scenario-Aware Data Flow
SDF	Synchronous Data Flow
SDR	Software Defined Radio
SDTC	Scheduling and Data Transfer Configuration
SysteMoC	SystemC Models of Computation

VF	Vectorization Factor
WSDF	Windowed Synchronous Data Flow

Contents

36.1 Introduction

Software design processes have evolved rapidly over the past two decades. In many areas, agile programming [1] has shown how software development benefits from going to implementation quickly. By writing core functionality for key use cases, software engineers can gain early feedback from real implementations, and, thereby, features, performance, and platforms may be refined effectively and quickly. Hardware/Software Codesign (HSCD) stands to inherit these same benefits from agile design but in practice has not kept pace with traditional software development evolution. Domain-specific models and languages that support fast application descriptions already exist. However, compared to traditional software, Hardware/Software tools to translate those descriptions to implementations are inherently more complex. They must deal with traditional software development issues, as well as system heterogeneity, multiple cores, and hardware accelerators. Because of the diversity of applicable tools and approaches, many of the steps are manual, ad hoc, or platform specific.

The purpose of this chapter is to discuss a Computer-Aided Design (CAD) framework for Digital Signal Processing (DSP) applications, called the DSPCAD Framework, that addresses some of these key development issues for the broad domain of DSP. The DSPCAD Framework achieves this by establishing a cross-platform, domain-specific approach that enables designers to arrive at initial implementations quickly for early feedback, and then systematically refine them toward functionally correct and high-performance solutions. The keys to such an approach include (a) lightweight design principles, which can be applied relatively quickly and flexibly in the context of existing design processes and (b) software techniques and tools that are grounded in data-flow models of computation.

36.1.1 Data Flow

Data-flow models have proven invaluable for DSP system design. Their graph-based formalisms allow designers to describe applications in a natural yet semantically rigorous way. As a result, data-flow languages are increasingly popular. Their diversity, portability, and intuitive design have extended them to many application areas and platform types within the broad DSP domain (e.g., see [3]). Modeling applications through coarse-grain data-flow graphs is widespread in the DSP design community, and a variety of data-flow models of computation have been developed for DSP system design.

Common to each of these modeling paradigms is the representation of computational behavior in terms of data-flow graphs. In this context of DSP system design, a data-flow graph is a directed graph $G = (V, E)$ in which each vertex (*actor*) $v \in V$ represents a computational task, and each edge $e \in E$ represents First-In First-Out (FIFO) communication of data values (*tokens*) from the actor $src(e)$ at the source of e to the actor $snk(e)$ at the sink of e. Data-flow actors execute in terms of discrete units of execution, called *firings*, which produce and consume tokens from the incident edges. When data-flow graphs are used for behavioral modeling of DSP systems, the graph represents application functionality with minimal details pertaining to implementation. For example, how the FIFO communication associated with each edge is mapped into and carried out through physical storage, and how the execution of the actors is coordinated are implementation-related details that are not part of the data-flow graph representation. Such orthogonalization between behavioral aspects and key implementation aspects is an important feature of data-flow-based DSP system design that can be leveraged in support of agile design processes. For a detailed and rigorous treatment of general principles of data-flow modeling for DSP system design, we refer the reader to [31], and for discussion on the utility of orthogonalization in system-level design, we refer the reader to [28].

36.1.2 Data-Flow Modeling Variants

A distinguishing aspect of data-flow modeling for DSP system design is the emphasis on characterizing the rates at which actors produce and consume tokens from their incident edges, and the wide variety of different variants of data-flow

models of computation that has evolved, due in large part to different assumptions and formulations involved in these *data-flow rates* (e.g., see [3, 49]). For example, Synchronous Data Flow (SDF) is a form of data flow in which each actor consumes a constant number of tokens from each input port and produces a constant number of tokens on each output port on every firing [30]. SDF can be viewed as an important common denominator that is supported in some fashion across most data-flow-based DSP design tools, and a wide variety of techniques for analyzing SDF graphs and deriving efficient implementations from them has been developed (e.g., see [3]). However, the restriction to constant-valued data-flow rates limits the applicability of the SDF model. This has led to the study of alternative data-flow models that provide more flexibility in specifying inter-actor communication. Examples of such models include Boolean Data Flow (BDF), Core Functional Data Flow (CFDF), Cyclo-Static Data Flow (CSDF), Multi-Dimensional Synchronous Data Flow (MDSDF), Parameterized Synchronous Data Flow (PSDF), Scenario-Aware Data Flow (SADF), and Windowed Synchronous Data Flow (WSDF) [6, 7, 9, 27, 34, 43, 51].

36.1.3 DSPCAD Framework

The DSPCAD Framework is a CAD framework that helps designers to apply the formalisms of the data-flow paradigm in DSP-oriented, HSCD processes. The DSPCAD Framework is specifically oriented toward flexible and efficient exploration of interactions and optimizations across different signal processing application areas (e.g., speech processing, specific wireless communication standards, cognitive radio, and medical image processing), alternative data-flow models of computation (e.g., Boolean Data Flow (BDF), Core Functional Data Flow (CFDF), etc., as listed in Sect. 36.1.2), and alternative target platforms along with their associated platform-based tools (e.g., field programmable gate arrays, graphics processing units, programmable digital signal processors, and low-power microcontrollers).

The DSPCAD Framework is based on three complementary subsystems, which respectively provide a domain-specific modeling environment for experimenting with alternative, DSP-oriented data-flow modeling techniques; a lightweight, cross-platform environment for implementing DSP applications as data-flow graphs; and a flexible project development tool that facilitates DSP system integration and validation using different kinds of platform-based development tools. These subsystems of the DSPCAD Framework are called, respectively, the Data-flow Interchange Format (DIF), LIghtweight Data-flow Environment (LIDE) and DSPCAD Integrative Command Line Environment (DICE). While DIF, LIDE, and DICE can be used independently as stand-alone tools, they offer significant synergy when applied together for HSCD. The DSPCAD Framework is defined by such integrated use of these three complementary tools.

In the remainder of this section, we provide brief overviews of DIF, LIDE, and DICE. We cover these tools in more detail in Sects. 36.3, 36.4, and 36.5, respectively. Then in Sect. 36.6, we demonstrate their integrated use in the DSPCAD Framework to develop a platform-specific data-flow framework for mapping SDF

graphs into Graphics Processing Unit (GPU) implementations. This case study is presented to concretely demonstrate the DSPCAD Framework and its capability to derive specialized data-flow tools based on specific data-flow modeling techniques and target platforms. In Sect. 36.7, we summarize the developments of this chapter and discuss ongoing directions of research in the DSPCAD Framework.

DIF – DIF provides application developers an approach to application specification and modeling that is founded in data-flow semantics, accommodates a wide range of specialized data-flow models of computation, and is tailored for DSP system design [21, 22].

DIF is comprised of a custom language that provides an integrated set of syntactic and semantic features that capture essential modeling information of DSP applications without over-specification. DIF also includes a software package for reading, analyzing, and optimizing applications described in the language. Additionally, DIF supports mixed-grain graph topologies and hierarchical design in specification of data-flow related, subsystem- and actor-specific information. The data-flow semantic specification is based on data-flow modeling theory and independent of any specialized design tool.

DIF serves as a natural design entry point for reasoning about a new application or class of applications and for experimenting with alternative approaches to modeling application functionality. LIDE and DICE complement these abstract modeling features of DIF by supporting data-flow-based implementations on specific platforms.

LIDE – LIDE is a flexible, lightweight design environment that allows designers to experiment with data-flow-based implementations directly on customized programmable platforms. LIDE is "lightweight" in the sense that it is based on a compact set of application programming interfaces that can be retargeted to different platforms and integrated into different design processes relatively easily.

LIDE contains libraries of data-flow graph elements ("gems"), as described in Sect. 36.1.1, and utilities that assist designers in modeling, simulating, and implementing DSP systems using formal data-flow techniques. Here, by gems, we mean actor and edge implementations. The libraries of data-flow gems (mostly actor implementations) contained in LIDE provide useful building blocks that can be used to construct signal processing applications and that can be used as examples that designers can adapt to create their own, customized LIDE actors.

Schedules for LIDE-based implementations can be created directly by designers using LIDE Application Programming Interfaces (APIs) or synthesized by DIF, decreasing the time to initial implementation. Refinements based on initial implementations may occur at the data-flow level (e.g., using DIF) or at the schedule implementation or gems level with LIDE, giving an application developer an opportunity to efficiently refine designs in terms of performance or functionality.

DICE – DICE is a package of utilities that facilitates efficient management of software projects. Key areas of emphasis in DICE are cross-platform operation, support for model-based design methodologies, support for projects that integrate

heterogeneous programming languages, and support for applying and integrating different kinds of design and testing methodologies. The package facilitates research and teaching of methods for implementation, testing, evolution, and revision of engineering software. The package is a foundation for developing experimental research software for techniques and tools in the area of DSP systems. The package is cross-platform, supporting Linux, Mac OS, Solaris, and Windows (equipped with Cygwin) platforms. By using LIDE along with DICE, designers can efficiently create and execute unit tests for user-designed actors.

36.2 Related Work

In this section, we review a number of representative data-flow-based tools that are applied to modeling, simulation, and synthesis of DSP systems. The intent in this review is not to be comprehensive but rather to provide a sampling of representative, research-oriented data-flow-based tools that are relevant to DSP system design. We also summarize distinguishing aspects of the DSPCAD Framework in relation to the state of the art in data-flow research for DSP. For broader and deeper coverage of different data-flow-based design tools and methodologies, we refer the reader to [3].

36.2.1 Representative Tools

Parallel and Real-time Embedded Executives Scheduling Method (PREESM) is an Eclipse-based code generation tool for signal processing systems [37,41]. PREESM provides architecture modeling and scheduling techniques for multi-core digital signal processors. In PREESM, applications are modeled as a hierarchical extension of SDF called an algorithm graph, while the targeted architectures are modeled as architecture graphs, which contain interconnections of abstracted processor cores, hardware coprocessors, and communication media. PREESM then takes the algorithm graph, architecture graph, and application parameters and constraints as its inputs to automatically generate software implementations on multi-core programmable digital signal processors.

The multi-processor scheduler in PREESM is based on the List and Fast Scheduling methods described by Kwok [29]. A randomized version of the List Scheduling method is first applied to return the best solution observed during a designer-determined amount of time. The obtained best solution can be applied directly for software synthesis or be used to initialize the population of a genetic algorithm for further optimization. The capabilities of PREESM are demonstrated, for example, by the rapid prototyping of a state-of-the-art computer vision application in [38].

SystemC Models of Computation (SysteMoC) is a SystemC-based library that facilitates data-flow-based HSCD for DSP systems. Actor design in SysteMoC is based on a model that includes a set of functions and an actor Finite-State Machine (FSM). The set of functions is partitioned into *actions*, which are used for data processing and *guards*, which are used to check for enabled transitions in

the actor FSM. In [19], an MPEG-4 decoder application is provided as a case study to demonstrate the capability of SysteMoC to support system synthesis as well as design space exploration for HSCD processes. For more details about SysteMoC, we refer the reader to ▸ Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign".

Cal Actor Language (CAL) is a data-flow programming language that can be applied to develop hardware and software implementations [11]. Like designs in SysteMoC, CAL programs incorporate an integration of data flow and state machine semantics. Actor specification in CAL includes actions, guards, port patterns, priorities, and transitions between actions. Thus, data-flow actor design in CAL is similar to that in SysteMoC and (as we will see in Sect. 36.4) LIDE in terms of an underlying, state-machine-integrated, data-flow model of computation. A major advance provided by CAL has been through its use in a recent MPEG standard for Reconfigurable Video Coding (RVC) [25].

36.2.2 Distinguishing Aspects of the DSPCAD Framework

Perhaps the most unique aspects of the DSPCAD Framework compared to other data-flow tools such as PREESM, SysteMoC, and CAL are the (1) emphasis on orthogonalization across three major dimensions in model-based DSP system design – abstract data-flow models, actor implementation languages, and integration with platform-specific design tools – and (2) support for a wide variety of different data-flow modelings styles. Feature 1 here is achieved in the DSPCAD Framework through the complementary objectives of DIF, LIDE, and DICE, respectively.

Support for Feature 2 in the DSPCAD Framework is threefold. First, DIF is agnostic to any particular data-flow model of computation and is designed to support a large and easily extensible variety of models. Second, LIDE is based on a highly expressive form of data-flow CFDF, which is useful as a common model for working with and integrating heterogeneous data-flow models of computation. This is because various specialized forms of data flow can be formulated as special cases of CFDF (e.g., see [44]). More details about CFDF are discussed in Sect. 36.3.1. Third, LIDE contains flexible support for parameterizing data-flow actors and manipulating actor and graph parameters dynamically. This capability is useful for experimenting with various parametric data-flow concepts, such as PSDF, and parameterized and interfaced data-flow [9] meta model, and the hierarchical reconfiguration methodologies developed in the Ptolemy project [35].

The DSPCAD Framework can be used in complementary ways with other DSP design environments, such as those described above. The modularity and specialized areas of emphasis within DIF, LIDE, and DICE make each of these component tools useful for integration with other design environments. For example, DIF has been employed as an intermediate representation to analyze CAL programs and derive statically schedulable regions from within dynamic data-flow specifications [16], and, in the PREESM project, the CFDF model of computation employed by

LIDE has been used to represent dynamic data-flow behavior for applying novel architectural models during design space exploration [39].

Although the DSPCAD Framework is not limited to any specific domain of signal processing applications, the components of the framework have been applied and demonstrated most extensively to date in the areas of wireless communications, wireless sensor networks, and embedded computer vision. For elaboration on HSCD topics in these latter two domains, we refer the reader to ▸ Chap. 38, "Wireless Sensor Networks" and ▸ Chap. 40, "Embedded Computer Vision" respectively.

36.3 Data-Flow Interchange Format Overview

DIF provides a model-based design environment for representing, analyzing, simulating, and synthesizing DSP systems. DIF focuses on data-flow graph modeling and analysis methods where the details of actors and edges of a graph are abstracted in the form of arbitrary actor and edge attributes. In particular, implementation details of actors and edges are not specified as part of DIF representations.

The DIF environment is composed of the DIF language and the DIF package. The DIF language is a design language for specifying mixed-grain data-flow models for DSP systems. The DIF package, a software package that is built around the DIF language, contains a large variety of data-flow graph analysis and transformation tools for DSP application models that are represented in DIF. More specifically, the DIF package provides tools for (1) representing DSP applications using various types of data-flow models, (2) analyzing and optimizing system designs using data-flow models, and (3) synthesizing software from data-flow graphs. The software synthesis capabilities of DIF assume that actor implementations are developed separately (outside of the DIF environment) and linked to their associated actor models as synthesis-related attributes, such as the names of the files that contain the actor implementation code.

Unlike most data-flow-based design environments, which are based on some forms of static data-flow model or other specialized forms of data flow, DIF is designed specifically to facilitate formal representation, interchange, and analysis of different kinds of data-flow models and to support an extensible family of both static and dynamic data-flow models. Models supported in the current version of DIF include SDF [30], CSDF [6], MDSDF [34], BDF [7], PSDF [2], and CFDF. DIF also provides various analysis, simulation, and synthesis tools for CFDF models and its specialized forms. As motivated in Sect. 36.2, CFDF is useful as a common model for working with and integrating heterogeneous data-flow models of computation [44], which makes it especially useful for the purposes of the DIF environment. Examples of data-flow tools within the DIF package are tools for CFDF functional simulation [43], SDF software synthesis for programmable digital signal processors [23], and quasi-static scheduling from dynamic data-flow specifications [16, 42]. Due to the important role of CFDF in DIF, we introduce background on CFDF in the following section.

36.3.1 Core Functional Data Flow

CFDF is a dynamic data-flow model of computation in which the behavior of an actor A is decomposed into a set of modes $modes(A)$. Each firing of A is associated with a specific mode in $modes(A)$. For each mode $m \in modes(A)$, the data-flow rates (numbers of tokens produced or consumed) for all actor ports are fixed. However, these rates can vary across different modes, which allows for the modeling of dynamic data-flow behavior.

When a CFDF actor A fires in a particular mode m, it produces and consumes data from its incident ports based on the constant production and consumption rates associated with m, and it also determines the *next mode* $z \in modes(A)$ for the actor, which is the mode that will be active during the next firing of A. The next mode may be determined statically as a property of each mode or may be data dependent. Combinations of data-dependent next mode determination and heterogeneous data-flow rates across different modes can be used to specify actors that have different kinds of dynamic data-flow characteristics.

A CFDF actor has associated with it two computational functions, called the *enable* and *invoke* functions of the actor. These functions provide standard interfaces for working with the actor in the context of a schedule for the enclosing data-flow graph. The enable function for a given actor A returns a Boolean value that indicates whether or not there is sufficient data on the input edges and sufficient empty space on the output edges to accommodate the firing of A in its next mode.

The invoke function of an actor, on the other hand, executes the actor according to its designated next mode and does so without any use of blocking reads or writes on actor ports – that is, data is consumed and produced without checking for availability of data or empty space, respectively. It is assumed that these checks will be performed (a) either statically, dynamically (using the enable method), or using a combination of static and dynamic techniques and (b) before the associated firings are dispatched with the invoke function. Thus, overhead or reduced predictability due to such checking need not be incurred during execution of the invoke function. This decomposition of actor functionality into distinct enable and invoke functions can be viewed as a formal separation of concerns between the checking of an actor's fireability conditions and execution of the core processing associated with a firing.

Various existing data-flow modeling techniques, including SDF, CSDF, and BDF, can be formulated as special cases of CFDF [44]. For further details on CFDF semantics, we refer the reader to [43, 44].

36.3.2 Reconfigurable Modulator Example

Here, we present a practical application as an example of CFDF modeling. Figure 36.1a shows a dynamically reconfigurable modulator application (*RMOD*) that supports multiple source rates and multiple Phase Shift Keying (PSK) and Quadrature Amplitude Modulation (QAM) schemes. Actor C reads two run-time

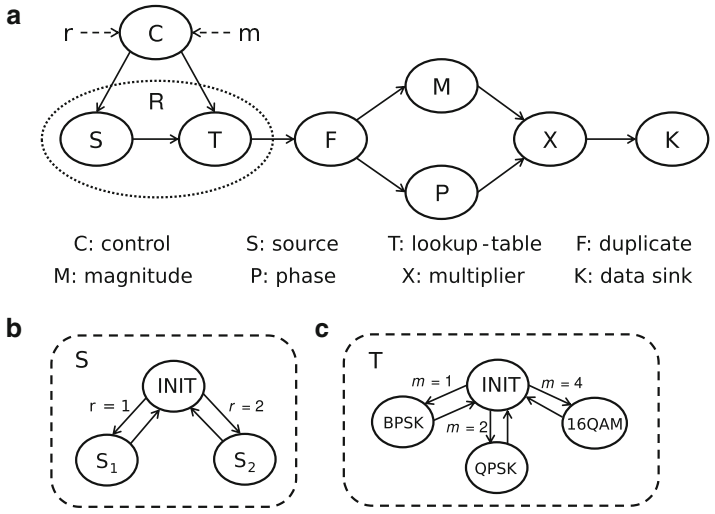


Fig. 36.1 CFDF modeling of a reconfigurable modulator (RMOD) application supporting multiple source data rate and modulation schemes. (**a**) CFDF model of the RMOD application (**b**) Mode transitions of actor S. (**c**) Mode transitions of actor T

a

Mode	Edge:C→S	Edge:S→T
INIT	–1	0
S_1	0	1
S_2	0	2

b

Mode	Edge:C→T	Edge:S→T	Edge:T→F
INIT	0	–1	0
BPSK	–1	0	1
QPSK	–2	0	1
16QAM	–4	0	1

Fig. 36.2 Data-flow tables. (**a**) Table for actor S. (**b**) Table for actor T

parameters, r and m, corresponding to the source data rate and modulation scheme, respectively, and sends these parameter values to the actors S and T. S and T in turn are two CFDF actors that each have multiple modes and data-dependent mode transitions, as illustrated in Figs. 36.1b and c, respectively.

Both S and T are initialized to begin execution in their respective INIT modes. In its INIT mode, S reads the source data rate r and switches to either S_1 or S_2 depending on the value of r. Similarly, in its INIT mode, T reads the modulation scheme index m and switches to one of the 3 modes, Binary PSK (BPSK), Quadrature PSK (QPSK), or 16-QAM, depending on m. S and T have different production and consumption rates in different modes.

Figure 36.2 shows the *data-flow tables* for actors S and T. A data-flow table Z for a CFDF actor A specifies the data-flow behavior for the available modes in the actor. Each entry $Z[\mu, p]$ corresponds to a mode $\mu \in modes(A)$ and input or output port p of A. If p is an output port of A, then $Z[\mu, p]$ gives the number of tokens produced on the edge connected to p during a firing of A in mode μ. Similarly, if

p is an input port, then $Z[\mu, p] = -c$, where c is the number of tokens consumed during mode μ from the edge connected to p.

In the column headings for the data-flow tables shown in Fig. 36.2, each port is represented by the edge that is connected to the port. If $m = 1$, then T executes in the BPSK mode and consumes only 1 token on its input edge. On the other hand, if $m = 4$, then T executes in the 16-QAM mode and consumes 4 tokens on its input edge. After firing in their respective BPSK or 16-QAM modes, S and T switch back to their INIT modes and await new values of r and m for the next round of computation. The remaining actors are SDF actors that consume/produce a single token on each of their input/output edges every time they fire.

36.3.3 Data-Flow Graph Specification in the DIF Language

As discussed above, the DIF language is a design language for specifying mixed-grain data-flow models in terms of a variety of different forms of data flow [22]. The DIF language provides a C-like, textual syntax for human-readable description of data-flow structure. An XML-based version of the DIF language, called *DIFML*, is also provided for structured exchange of data-flow graph information between different tools and formats [17]. DIF is based on a block-structured syntax and allows specifications to be modularized across multiple files through integration with the C preprocessor. As an example, a DIF specification of the RMOD application is shown in Listing 1.

Listing 1 DIF Language specification of the RMOD application

```
CFDF RMOD {
  topology {
    nodes = C, S, T, F, M, P, X, K;
    edges = e1(C, S), e2(C, T), e3(S, T), e4(T, F),
      e5(F, M), e6(F, P), e7(M, X), e8(P, X), e9(X, K);
  }
  actor C {
    name = "mod_ctrl";
    out_r = e1; out_m = e2; /* Assign edges to ports */
  }
  actor S {
    name = "mod_src";
    in_ctrl = e1; out_data = e3;
    mode_count = 3;
  }
  actor T {
    name = "mod_lut";
    in_ctrl = e1; in_bits = e3; out_symbol = e4;
    mode_count = 4;
  }
  /* Other actor definitions */
  /* ... */
}
```

In this example, the RMOD application is described using CFDF semantics, which is represented by the `cfdf` keyword in DIF. The `topology` block defines the actors (nodes) and edges of the data-flow graph and associates a unique identifier with each actor and each edge. Because data-flow graphs are directed graphs, each edge is represented as an ordered pair (u, v), where u is the source actor and v is the sink actor. Each actor can be associated with an optional `actor` block, where attributes associated with the actor are defined. The attributes can provide arbitrary information associated with the actor using a combination of *built-in* and *user-defined* attribute specifiers. In the example of Listing 1, the `actor` block specifies the following attributes: (1) the name of the implementation associated with the actor (to help differentiate between alternative implementations for the same abstract actor model), (2) input/output port connections with the incident edges, and (3) the number of CFDF modes for the actor.

In addition to the language features illustrated in Listing 1, DIF also supports a variety of other features for specifying information pertaining to data-flow-based application models. For example, DIF supports hierarchical specification, where an actor in one graph can be linked with a "nested" subgraph to promote top-down decomposition of complex graphical models and to help support different forms of semantic hierarchy, such as those involved in parameterized data-flow semantics [2]. Another feature in DIF is support for *topological patterns*, which enable compact, parameterized descriptions of various kinds of graphical patterns (e.g., chain, ring, and butterfly patterns) for instantiating and connecting actors and edges [46].

36.3.4 Model-Based Design and Integration Using DIF

The DIF package provides an integrated set of models and methods, illustrated in Fig. 36.3, for developing customized data-flow-model-based design flows targeted to different areas of signal processing, and different kinds of target platforms. As opposed to being developed primarily as a stand-alone data-flow tool, DIF is designed for flexibility in integrating established or novel data-flow capabilities into arbitrary model-based design environments for DSP. For example, Zaki presents a DIF-based tool for mapping Software Defined Radio (SDR) applications into GPU implementations, and integrating the derived mapping solutions into GNU Radio, which is a widely used environment for SDR system design [52]. As another example, DIF has been integrated to provide data-flow analysis and transformation capabilities for the popular data-flow language called CAL, which was discussed previously in Sect. 36.2.1. For details on this application of DIF to CAL, we refer the reader to [16, 17], while readers can find details about the CAL language in [10].

The DIF package consists of three major parts: the DIF representation, DIF-based graph analysis and transformation techniques, and tools for simulation and software synthesis.

DIF representation. The DIF package provides an extensible set of data structures that represent data-flow-based application models, as they are specified in the DIF language and as they are transformed into alternative models for the purposes of

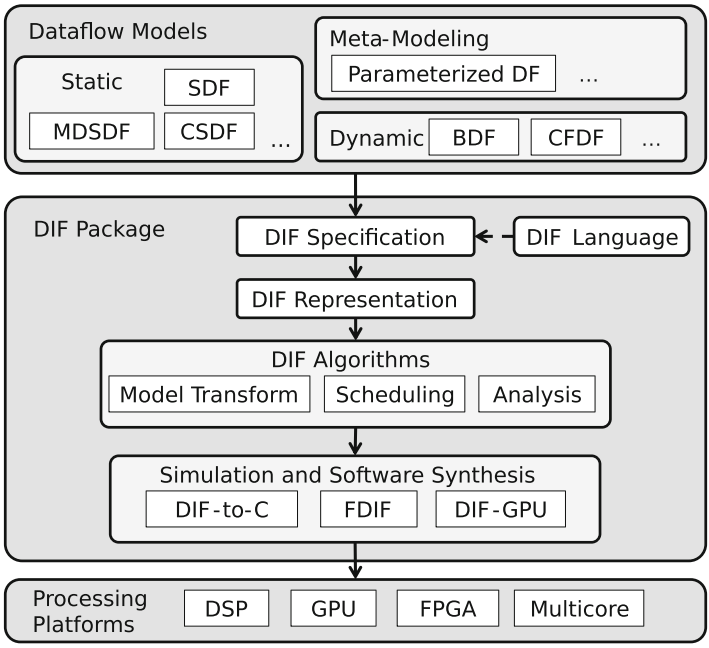


Fig. 36.3 Overview of the DIF package

analysis, optimization, or software synthesis. These graphical data structures are collectively referred to as the DIF intermediate representation or "DIF representation" for short. The initial DIF representation (before any transformations are applied) for a given DIF language specification is constructed by the DIF front-end tools, which are centered on a Java-based parser. This parser is developed using the SableCC compiler framework [13].

Analysis and Transformation Techniques. The DIF package provides implementations of a large set of methods for data-flow model analysis and transformation, including methods for scheduling, and buffer management. These methods operate on the graphical data structures within the DIF representation. The analysis and transformation techniques provided in DIF are useful in many aspects of data-flow-based design and implementation.

Simulation and Software Synthesis. DIF presently includes a number of tools for simulation and software synthesis from data-flow models. Functional DIF (FDIF) simulates CFDF-based models where actor functionality is programmed in terms of CFDF semantics using Java [43] along with CFDF-specific APIs. FDIF is designed especially to help designers to efficiently prototype and validate alternative kinds of static, dynamic, and quasi-static scheduling strategies. The DIF-to-C tool generates C code that is optimized for efficient execution on programmable digital signal processors [23]. The software synthesis capabilities in DIF-to-C are integrated with a variety of analysis and transformation techniques in DIF so that designers

can apply different combinations of transformations to explore trade-offs among data memory requirements, code size, and execution speed. DIF-GPU is a newly developed software synthesis tool that is targeted to heterogeneous CPU–GPU platforms. Currently, DIF-GPU generates multi-threaded Compute Unified Device Architecture (CUDA) application code that can utilize both Central Processing Units (CPUs) and GPUs for implementation of high-performance DSP systems. Further details on DIF-GPU are discussed in Sect. 36.6.

36.4 Lightweight Data-Flow Environment

LIDE facilitates design and implementation of DSP actors and systems using a structured, CFDF-based data-flow approach that can be integrated with a wide variety of platform-oriented languages, such as C, CUDA, OpenCL, Verilog, and VHDL [47, 48]. LIDE is centered on a compact set of abstract APIs for developing data-flow actors and edges. These APIs are (1) defined in terms of fundamental data-flow principles, (2) independent of any specific programming language, and (3) readily retargetable across a wide variety of specific languages for DSP implementation, including the platform-oriented languages listed above.

LIDE is designed with a primary objective of allowing DSP system designers to apply and experiment with data-flow techniques relatively easily in the context of their existing design processes, language preferences, and target platforms. This objective is supported by the compact collection of retargetable, language-agnostic APIs that LIDE is based on. LIDE also provides collections of pre-designed data-flow gems, as described in Sect. 36.1.3.

When LIDE is integrated with a specific programming language XYZ for implementing gems, we refer to the resulting integrated design tool as LIDE-XYZ or in some cases as LIDE-X if X is used as an abbreviation for XYZ. Existing subsystems within LIDE include LIDE-C, LIDE-CUDA, LIDE-V, and LIDE-OCL, where the latter two represent the integration of LIDE with the Verilog Hardware Description Language (HDL) and OpenCL, respectively.

36.4.1 Actor Design in LIDE

As described previously, actor implementation in LIDE is based on the CFDF model of computation. This choice of CFDF as the modeling foundation for LIDE is motivated by the high expressive power of CFDF, and its utility in working with heterogeneous forms of data flow [44].

Actor design in LIDE includes four basic interface functions, which are referred to as the construct, enable, invoke, and terminate functions of an actor. The *construct* function instantiates an actor and performs pre-execution initialization of the actor, such as initializing values of actor parameters and allocating storage that is related to the state of the actor. Conversely, the *terminate* function performs any operations that are required for "closing out" the actor after the enclosing graph has finished

executing. Such operations include freeing memory that has been allocated in the corresponding construct function.

The *enable* and *invoke* functions provide direct interfaces for key concepts of CFDF semantics, which were discussed in Sect. 36.3.1. As one would guess, the enable and invoke functions in LIDE are defined to provide implementations for the enable and invoke functions in CFDF semantics. We employ a minor abuse of terminology here, where this pair of functions is defined with the same names in both LIDE (a design tool) and CFDF (a model of computation). Where there may be confusion, one may qualify a reference to the function with an appropriate reference to the tool or model (e.g., "the LIDE enable function").

The enable and invoke functions in LIDE provide flexible interfaces for implementing arbitrary schedulers, including static, dynamic, and quasi-static schedulers, for executing data-flow graph implementations. The enable function is implemented by the actor programmer to check whether or not the actor has sufficient tokens on its input ports and enough empty space on its output ports to support a single firing in next CFDF mode of execution that is currently associated with the actor. Similarly, the invoke function is implemented to execute a single firing of the actor according to its next mode. The invoke function should also update the next mode of the actor, which in turn determines the conditions that will be checked by the enable function if it is called prior to the next actor firing.

When the invoke function is called, it is assumed that sufficient input tokens and output space are available (since there is a separate API function dedicated to checking these conditions). Thus, the actor programmer should not implement checks for these conditions within the invoke function. These conditions should be satisfied – as part of the design rules of any tool that implements CFDF semantics – before calling the invoke function to execute a given actor firing.

We emphasize that in a given scheduler for an enclosing data-flow graph, it is not always necessary to call the enable function of an actor before calling the invoke function. In particular, such calls to the enable function can be bypassed at run time if the corresponding conditions are guaranteed through other forms of analysis, including any combination of static, dynamic, and hybrid static/dynamic analysis. For example, when implementing the scheduler for a LIDE-based data-flow graph that consists only of SDF or CSDF actors, the use of the enable function can be avoided entirely if a static schedule is employed [6, 30]. This allows designers in LIDE to more effectively utilize the large collection of available static scheduling techniques for SDF and CSDF representations (e.g., see [3, 8, 12, 14, 36, 40, 45]).

For more details on actor implementation in LIDE, we refer the reader to [48].

36.4.2 Parameterized Sets of Modes

Actor design in LIDE naturally supports the concept of Parameterized Sets of Modes (PSM), which is a modeling enhancement to CFDF that enables designers to more concisely specify and work with actor behavior that involves groups of related modes [33].

For example, consider an actor A that has two input ports *in1* and *in2*, and a single output port *out*. The actor starts execution in a mode called *read_length*, which consumes a single, positive integer-valued token from *in1*, and stores this consumed value in a state variable N. The value of the input token consumed from *in1* is restricted to fall in the range $\{1, 2, \ldots, M\}$, where M is a parameter of A. In the next firing, the actor consumes a vector spanning N input tokens from *in2*, computes the maximum of these N values, outputs the result (as a single token) on *out*, and determines its next mode to be the *read_length* mode. Thus, intuitively, the actor executes through alternate firings where (a) a vector length is read and used to determine the consumption rate of a subsequent mode, and then in this subsequent mode, (b) a vector is read and processed to produce a single output token.

Using standard CFDF notation, we can represent this as an actor that has $(M+1)$ distinct modes, i.e., as M different "vector processing modes" in addition to the *read_length* mode. However, such a representation can become unwieldy, especially if M is large. A PSM is a level of abstraction that allows us to group together a collection of related modes with one or more parameters that are used to select a unique mode from the collection at run time. These parameters can be determined statically or dynamically, allowing for significant flexibility in how PSMs are applied to actor design.

In this simple vector processing example, the M vector processing modes can be grouped together into a single PSM *vect_proc*, and with an associated parameter *vect_len* whose value corresponds to the value of the actor state variable N.

Technically, an actor *mode* in LIDE corresponds to a PSM rather than an individual CFDF actor mode. A LIDE actor can produce or consume different numbers of tokens in the same mode as long as the data-flow rates are all uniquely determined by the LIDE actor mode PSM and the values of the actor parameters that are associated with that PSM. Such unique determination of data-flow rates ensures that the underlying actor behavior corresponds to CFDF semantics, while allowing the code to be developed and the actor functionality to be reasoned about in terms of the higher-level PSM abstraction.

For a more formal and thorough development of PSM-based modeling, we refer the reader to [33].

36.4.3 Implementation in LIDE

In this section, we discuss details of design and implementation of data-flow components in LIDE using an example based on LIDE-C. In LIDE-C, data-flow gems are implemented in the C language. A collection of gems and utilities is provided as part of LIDE-C. These can be linked through various LIDE-C libraries into data-flow graph implementations, and they can also serve as useful templates or examples to help users develop new gems for their applications.

More specifically, LIDE-C contains a set of libraries called `gems`, and another library called `tools`. Basic actor and edge FIFO implementations are provided in

gems, while basic utilities, including a simple scheduler, are accessible in tools. The scheduler provided in LIDE-C is a basic form of CFDF scheduler, called a *canonical scheduler* [44]. This form of scheduler can be applied to arbitrary data-flow graphs in LIDE-C. Because it is general and easy to use, it is useful for functional validation and rapid prototyping. However, it is relatively inefficient, as it is designed for simplicity and generality, rather than for efficiency.

More efficient schedulers can be implemented by LIDE-C designers using the core LIDE APIs, including the enable and invoke functions for the actors. Each LIDE-C actor, as a concrete form of LIDE actor, must have implementations of these functions. LIDE-C schedulers can also be generated automatically through software synthesis tools.

36.4.3.1 Data-Flow Graph Components

In LIDE-C, gems (actors and FIFOs) are implemented as Abstract Data Types (ADTs) in C. Such ADT-based implementation provides a C-based, object-oriented design approach for actors and FIFOs in LIDE-C. As we discussed in Sect. 36.4.1, each LIDE actor has four standard interface functions. The developer of an actor in LIDE-C must provide implementations of these functions as methods – referred to as the new, enable, invoke, and terminate methods – of the ADT for the actor.

An analogous process is followed for FIFO design in LIDE-C and in related targets of LIDE, including LIDE-CUDA and LIDE-OCL. In particular, users can define any number of different FIFO types (e.g., corresponding to different forms of physical implementation, such as mappings to different kinds of memories), where each FIFO type is designed as an ADT. For example, in LIDE-OCL, which is currently developed for hybrid CPU-GPU implementation platforms, two FIFO ADTs are available – one for implementation of the FIFO on a CPU and another for implementation on a GPU.

The abstract (language-agnostic) LIDE API contains a set of required interface functions for FIFOs that implement edges in LIDE programs. In LIDE-C, FIFOs are implemented as ADTs where the required interface functions are implemented as methods of these ADTs. Required interface functions for FIFOs in LIDE include functions for construction and termination (analogous to the construct and terminate functions for actors), reading (consuming) tokens, writing (producing) tokens, querying the number of tokens that currently reside in a FIFO, and querying the capacity of a FIFO. The capacity of a FIFO in LIDE is specified through an argument to the construct function of the FIFO.

Listing 2 shows the function prototypes for the new, enable, invoke, and terminate methods in LIDE-C. In addition to these interface functions, designers can add auxiliary functions in their actor implementations. For working with actor parameters, components in the LIDE-C libraries employ a common convention of using corresponding set and get methods associated with each parameter (e.g., set_tap_count, get_tap_count).

Listing 2 The format for function prototypes of the new, enable, invoke, and terminate methods of a LIDE-C actor

```
lide_c_<actor_name>_context_type *
    lide_c_<actor_name>_new([FIFO pointer list],
    [parameter list])

boolean lide_c_<actor_name>_enable(
    lide_c_<actor_name>_context_type *context)

void lide_c_<actor_name>_invoke(
    lide_c_<actor_name>_context_type *context)

void lide_c_<actor_name>_terminate(
    lide_c_<actor_name>_context_type *context)
```

Each function prototype shown in Listing 2 involves an argument that points to a data structure that is referred to as the *actor context* (or simply "context"). Each actor A in a LIDE-C data-flow graph implementation has an associated context, which encapsulates pointers to the FIFOs that are associated with the edges incident to A; function pointers to the `enable` and `invoke` methods of A; an integer variable that stores the index of the current CFDF mode or PSM of A; and parameters and state variables of A.

For purposes of data-flow graph analysis or transformation (e.g., as provided by DIF), a LIDE actor that employs one or more state variables can be represented by attaching a self-loop edge to the graph vertex associated with the actor. Here, by a *self-loop edge*, we mean an edge e for which $src(e) = snk(e)$. In general, one can also use such a self-loop edge to represent inter-firing dependencies for an actor that can transition across multiple CFDF modes at run time (here, the mode variable acts as an implicit state variable). On the other hand, if the mode is uniquely determined at graph configuration time and does not change dynamically, then this "CFDF-induced" self-loop edge can be omitted. Such an actor can, for example, be executed in a data parallel style (multiple firings of the actor executed simultaneously) if there are no state-induced self-loop edges or other kinds of cyclic paths in the data-flow graph that contain the actor.

36.4.3.2 Actor Implementation Example

As a concrete example of applying LIDE-C, we introduce in this section a LIDE-C implementation of a modulation selection actor. Recall that such an actor is employed as actor T in the RMOD application that was introduced in Sect. 36.3.2. This actor T is an example of CFDF semantics augmented with the concept of PSMs. Recall that the data-flow graph and data-flow tables for this actor are shown in Figs. 36.1 and 36.2.

Listing 3 and Listing 4 illustrate key code segments within the `enable` and `invoke` methods, respectively, for actor T in our LIDE-C implementation of the actor. These code segments involve carrying out the core computations for determining fireability and firing the actor, respectively, based on the actor mode

or PSM that is active when the corresponding method is called. For conciseness, each of these two listings shows in detail the functionality corresponding to a single mode, along with the overall structure for selecting an appropriate action based on the current mode (using a `switch` statement in each case). Lines marked with "`.....`" represent code that is omitted from the illustrations for conciseness. Interface functions whose names start with `lide_c_fifo` are methods of a basic FIFO ADT that is available as part of LIDE-C.

Listing 3 Code within the enable method for actor T in the RMOD application

```
/* context: structure that stores actor information. E.g.,
   context->mode stores the actor's current mode.
*/
switch (context->mode) {
    case LIDE_C_RMOD_T_MODE_INIT:
        result = (lide_c_fifo_population(context->fifo_ctrl_input)
                    >= 1);
        break;
    case LIDE_C_RMOD_T_MODE_BPSK:
        result = ......
        break;
    case LIDE_C_RMOD_T_MODE_QPSK:
        result = ......
        break;
     case LIDE_C_RMOD_T_MODE_QAM16:
        result = ......
        break;
    default:
        result = FALSE;
        break;
}
return results;
```

Listing 4 Code within the invoke method for actor T in the RMOD application

```
switch (context->mode) {
     case LIDE_C_RMOD_T_MODE_INIT:
        /* scheme: variable indicating BPSK, QPSK or QAM16 */
        lide_c_fifo_read(context->fifo_ctrl_input, &scheme);
        context->mode = scheme;
        /* nbits: number of bits to process for the given scheme
           rb: remaining bits before switching scheme */
        context->rb = context->nbits;
        break;
     case LIDE_C_RMOD_T_MODE_BPSK:
        lide_c_fifo_read_block(context->fifo_data_input,
                               &bits, 1);
        code.x = context->bpsk_table[bits].x;
        code.y = context->bpsk_table[bits].y;
        lide_c_fifo_write(context->fifo_data_output, &code);
        context->rb --;
```

```
        if (context->rb > 0) {
            context->mode = LIDE_C_RMOD_T_MODE_BPSK;
        } else {
            context->mode = LIDE_C_RMOD_T_MODE_INIT;
        }
        break;
    case LIDE_C_RMOD_T_MODE_QPSK:
        ......
        break;
    case LIDE_C_RMOD_T_MODE_QAM16:
        ......
        break;
   default:
        context->mode = LIDE_C_RMOD_T_MODE_INACTIVE;
        break;
}
```

36.5 DSPCAD Integrative Command Line Environment

In this section, we describe the DSPCAD Integrative Command Line Environment (DICE), which is a Bash-based software package for cross-platform and model-based design, implementation, and testing of signal processing systems [5]. The DICE package is developed as part of the DSPCAD Framework to facilitate exploratory research, design, implementation, and testing of digital hardware and embedded software for DSP. DICE has also been used extensively in teaching of cross-platform design and testing methods for embedded systems (e.g., see [4]). DICE has been employed to develop research prototypes of signal processing applications involving a wide variety of platforms, including desktop multi-core processors, Field-Programmable Gate Arrays (FPGAs), GPUs, hybrid CPU-GPU platforms, low-power microcontrollers, programmable digital signal processors, and multi-core smartphone platforms. An overview of DICE is given in [5], and an early case study demonstrating the application of DICE to DSP system design is presented in [26]. In the remainder of this section, we highlight some of the most complementary features of DICE in relation to LIDE and DIF.

Because DICE is based on Bash, it has various advantages that complement the advantages of platform-based or language-specific integrated development environments (IDEs). For example, DICE can be deployed easily on diverse operating systems, including Android, Linux, Mac, Solaris, and Windows (with Cygwin). The primary requirement is that the host environment should have a Bash command line environment installed. DICE is also agnostic to any particular actor implementation language or target embedded platform. This feature of DICE helps to provide a consistent development environment for designers, which is particularly useful when developers are experimenting with diverse hardware platforms and actor implementation languages.

36.5.1 Convenience Utilities

DICE includes a collection of simple utilities that facilitate efficient directory navigation through directory hierarchies. This capability is useful for working with complex, cross-platform design projects that involve many layers of design decomposition, diverse programming languages, or alternative design versions for different subsystems. These directory navigation operations help designers to move flexibly and quickly across arbitrary directories without having to traverse through multiple windows, execute sequences of multiple `cd` commands, or type long directory paths. These operations also provide a common interface for accelerating fundamental operations that is easy to learn and can help to quickly orient new members in project teams.

DICE also provides a collection of utilities, called the Moving Things Around (MTA) utilities, for easily moving or copying files and directories across different directories. Such moving and copying is common when working with design projects (e.g., to work with code or documentation templates that need to be copied and then adapted) and benefit from having a simple, streamlined set of utilities. The MTA utilities in DICE are especially useful when used in conjunction with the directory navigation utilities, described above.

Some of the key directory navigation utilities and MTA utilities in DICE are summarized briefly in Table 36.1.

The items in Table 36.1 that are enclosed in angle brackets (`<...>`) represent placeholders for command arguments. The abbreviation-based names of the first three utilities listed in Table 36.1 are derived as follows: `dlk` stands for (create) Directory LinK, `g` stands for Go, and `rlk` stands for Remove LinK. The other two utilities listed in Table 36.1 use a naming convention that applies to many core utilities in DICE where the prefix "dx" is used at the beginning of the utility name. The name `dxco` stands for (CO)py (a file or directory), and `dxparl` stands for paste and remove the last file or directory transferred.

Table 36.1 Selected navigation utilities and MTA utilities in DICE

Utility	Description
`dlk <label>`	Associate the specified label with the Current Working Directory (CWD)
`g <label>`	Change directory to the directory that is associated with the specified label
`rlk <label>`	Remove the specified label from the set of available directory navigation labels
`dxco <arg>`	Copy the specified file or directory to the DICE user clipboard
`dxparl`	Paste (copy) into the CWD the last (most recent) file or directory that has been transferred to the to the DICE user clipboard, and remove this file or directory from the clipboard

36.5.2 Testing Support

One of the most useful sets of features in DICE is provided by its lightweight and language-agnostic unit testing framework. This framework can be applied flexibly across arbitrary actor implementation languages (C, CUDA, C++, Java, Verilog, VHDL, etc.) and requires minimal learning of new syntax or specialized languages [26]. The language-agnostic orientation of DICE is useful in heterogeneous development environments, including codesign environments, so that a common framework can be used to test across all of the relevant platforms.

In a DICE-based test suite, each specific test for an HDL or software implementation unit is implemented in a separate directory, called an Individual Test Subdirectory (ITS), which is organized in a certain way according to the DICE-based conventions for test implementation. To be processed by the DICE facilities for automated test suite execution, the name of an ITS must begin with `test` (e.g., `test01`, `test02`, `test-A`, `test-B`, `test_square_matrix`). To exclude a test from test suite evaluation, one can simply change its name so that it does not begin with `test`.

36.5.2.1 Required Components of an ITS

Here, we describe the required components of an ITS. Except for the set of input files for the test, each of these components takes the form of a separate file. The set of input files may be empty (no files) or may contain any number of files with any names that do not conflict with the names of the required ITS files, as listed below:

- A file called `test-desc.txt` that provides a brief explanation of what is being tested by the ITS, that is, what is distinguishing about this test compared to the other tests in the test suite.
- An executable file (e.g., some sort of script) called `makeme` that performs all necessary compilation steps (e.g., compilation of driver programs) that are needed for the ITS. Note that the compilation steps performed in the `makeme` file for a test typically do *not* include compilation of the source code that is being tested; project source code is assumed to be compiled separately before a test suite associated with the project is exercised.
- An executable file called `runme` that runs the test and directs all normal output to standard output, and all error output to standard error.
- Any input files that are needed for the test.
- A file called `correct-output.txt` that contains the standard output text that should result from the test. If no output is expected on standard output, then `correct-output.txt` should exist in the ITS as an empty file.
- A file called `expected-errors.txt` that contains the standard error text that should result from the test. This placeholder provides a mechanism to test the correct operation of error detection and error reporting functionality. If no

output is expected on standard error, then `expected-errors.txt` should exist in the ITS as an empty file.

The organization of an ITS is structured in this same, language-independent, form – based on the required items listed above – regardless of how many and what kinds of design languages are involved in a specific test. This provides many benefits in DSP codesign, where several different languages and back-end (platform-based) tools may be employed or experimented with in a given system design. For example, the DICE test suite organization allows a designer or testing specialist to switch between languages or project subsystems without being distracted by language-specific peculiarities of the basic structure of tests and their operation.

As one might expect from this description of required files in an ITS, a DICE-based test is evaluated by automatically comparing the standard output and standard error text that is generated by `runme` to the corresponding `correct-output.txt` and `expected-errors.txt` files.

Note that because of the configurable `runme` interface, it is not necessary for all of the output produced by the project code under test to be treated directly as test output. Instead, the `runme` script can serve as a wrapper to filter or reorganize the output generated by a test in a form that the user finds most efficient or convenient for test management. This provides great flexibility in how test output is defined and managed.

36.5.2.2 Relationship to Other Testing Frameworks and Methodologies

The DICE features for unit testing are largely complementary to the wide variety of language-specific testing environments (e.g., see [18, 24, 50]). More than just syntactic customizations, such frameworks are often tied to fundamental constructs of the language. DICE can be used to structure, organize, and execute in a uniform manner unit tests that employ language-specific and other forms of specialized testing frameworks. For example, specialized testing libraries for Java in a simulation model of a design can be employed by linking the libraries as part of the `makeme` scripts in the ITSs of that simulation model. When a designer who works primarily on hardware implementation for the same project examines such a "simulation ITS," he or she can immediately understand the overall organization of the associated unit test and execute the ITS without needing to understand the specialized, simulation-specific testing features that are employed.

DICE is also not specific to any specific methodology for creating or automatically generating unit tests. A wide variety of concepts and methods have been developed for test construction and generation (e.g., see [20]). By providing a simple and flexible environment for implementing, executing, and managing tests, the DICE unit testing framework can be used to prototype different kinds of test development methodologies and apply them in arbitrary implementation contexts.

For further details on the process of test implementation in DICE, and the relationship of DICE to other testing frameworks, we refer the reader to [4, 5, 26].

36.6 DSPCAD Framework Example: DIF-GPU

In this section, we demonstrate the DSPCAD Framework by describing its use to develop DIF-GPU, a software synthesis tool for mapping SDF graphs onto hybrid CPU-GPU platforms. Using DIF-GPU, a DSP designer can specify a signal-flow graph as an SDF graph in the DIF language; implement the individual actors of the graph in LIDE-CUDA; automatically schedule and generate interacting CPU and GPU code for the graph; and validate the generated implementation using the cross-platform testing capabilities of DICE.

We note that the case study presented in this section is not intended to emphasize details of a specific data-flow tool for GPU implementation but rather to demonstrate how the complementary resources and capabilities in the DSPCAD Framework can be applied to efficiently prototype such a tool. For a detailed presentation of the DIF-GPU tool, we refer the reader to [32].

36.6.1 DIF-GPU Overview

DIF-GPU targets heterogeneous CPU-GPU platforms in which multi-core CPUs and GPUs operate concurrently to provide high-performance signal processing capability. Modern GPUs can contain hundreds or thousands of single instruction multiple data (SIMD) multi-processor cores to process large amounts of data in parallel. Such an architecture enables GPUs to obtain significant performance gain over CPUs on data parallel tasks. Cooperation between a multi-core CPU and GPU allows various types of parallelism to be exploited for performance enhancement, including pipeline, data, and task parallelism.

DIF-GPU targets CPU-GPU platforms that are modeled as host-device architectures where the CPU is referred to as the "host" and the GPU as the "device," and where the employed CPUs, main memory, and GPUs are connected by a shared bus. CPUs control the GPUs by dispatching commands and data from main memory, while GPUs perform their assigned computations in their local memories (device memory). A GPU's device memory is private to that GPU and separated from main memory and the memories of other devices. Data transfers between the host and individual devices are referred to as Host-to-Device (H2D) or Device-to-Host (D2H) data transfers, depending on the direction. H2D and D2H data transfers can produce large overhead and significantly reduce the performance gain provided by GPUs (e.g., see [15]). To achieve efficient implementations in DIF-GPU, such overhead is taken carefully into account in the processes of task scheduling and software synthesis.

DIF-GPU is developed using the integrated toolset of the DSPCAD Framework, including DIF, LIDE, and DICE. Methods for data-flow analysis, transformation, scheduling, and code generation are developed by building on capabilities of the DIF package. Implementation of GPU-accelerated actors and run time, multi-threaded execution support are developed by applying LIDE-CUDA. Unit testing and application verification are carried out using DICE.

36.6.2 Graph Transformations and Scheduling using DIF

Figure 36.4 illustrates the overall workflow of DIF-GPU. This workflow consists of 3 major steps, vectorization, Scheduling and Data Transfer Configuration (SDTC), and code generation. Data parallelism is exploited by the vectorization step, while pipeline and task parallelism are exploited by the SDTC step.

36.6.3 Vectorization

Data-flow graph vectorization can be viewed as a graph transformation that groups together multiple firings of a given actor into a single unit of execution [45]. The number of firings involved in such a group is referred to as the Vectorization Factor (VF). Vectorization is a useful method for exploiting data parallelism in data-flow models.

Suppose that A is an actor in an SDF graph G, and G' represents the transformed graph that results from replacing A with a vectorized version A_b of A with $VF = b$. The edges in G' are the same as those in G, except that for all input edges of A_b, the consumption rates are effectively multiplied by b (relative to their corresponding rates in G), and similarly, for all output edges of A_b, the production rates are multiplied by b.

Vectorization exposes potential for exploiting parallelism across multiple firings of the same actor. For example, when executing A_b on a GPU, blocks of b firings of A can be executed together concurrently on stream processors in the GPU.

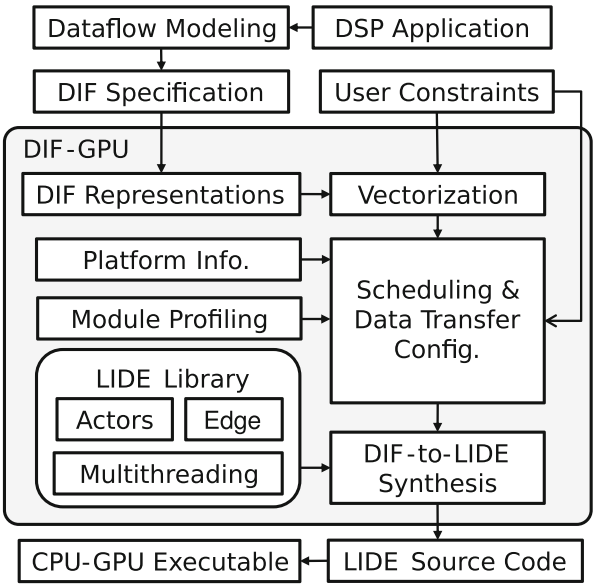


Fig. 36.4 The DIF-GPU workflow

DIF-GPU applies vectorization in a form called Graph-Level Vectorization (GLV) [52] for SDF graphs. GLV involves a positive integer parameter J that is called the *GLV degree* for the input SDF graph G. In GLV, J iterations of a minimal periodic schedule for G are scheduled together, and the GLV degree is used in conjunction with the repetitions vector q for G to derive the VF for each actor. For background on periodic schedules and repetitions vectors for SDF graphs, we refer the reader to [30].

More specifically, in GLV, the VF for each actor A is derived as $J \times q(A)$, where $q(A)$ represents the repetitions vector component that is indexed by A. After transforming each actor by its VF associated with a given GLV degree J, the resulting vectorized graph G_{vect}, is a *single-rate* SDF graph that represents the execution of J successive iterations of a minimal periodic schedule for G. Here, by a single-rate SDF graph, we mean that the repetitions vector components are uniformly equal to unity – that is, if r represents the repetitions vector for G_{vect}, then for every actor A in G_{vect}, $r(A) = 1$.

In DIF-GPU the input SDF graph is assumed to be acyclic (apart from the possibility of self-loop edges induced by actor state) so that there are no cyclic paths in the application graph that impose limitations on the GLV degree. A wide variety of practical signal processing systems can be represented in the form of acyclic SDF graphs (e.g., see [3]). The techniques employed in DIF-GPU can readily be extended to more general graph topologies, e.g., by applying them outside of the strongly connected components of the graphs. Such an extension is a useful direction for further development in DIF-GPU.

Actors in DIF-GPU are programmed using a VF parameter, which becomes part of the actor context in LIDE-CUDA. The actor developer implements vectorized code for each actor in a manner that is parameterized by the associated VF parameter and that takes into account any limitations in data parallel operation or memory management constraints imposed by actor state. For example, to implement a vectorized Finite Impulse Response (FIR) filter in DIF-GPU, a VF parameter is included in the associated LIDE-C actor context such that the actor consumes and produces VF tokens in each firing. Along with this VF parameter, the actor context contains pointers to (1) an array of filter coefficients and (2) an array of past samples for the filter. The past samples array, which contains $(N - 1)$ elements, stores the most recently consumed $(N - 1)$ tokens by the actor. Here, N is the order of the filter. Firing the vectorized GLV filter involves consuming b input tokens, generating b output tokens, and updating the actor state that is maintained in the past samples array, where b is the value of the VF parameter. Using careful buffer management within the LIDE-CUDA actor implementation, the b output samples for the actor are computed in parallel on the target GPU assuming that there are sufficient resources available in the GPU in relation to N and b.

The GLV approach employed in DIF-GPU is useful because it provides a single parameter (the GLV degree) that can be employed to control system-level trade-offs associated with vectorization and thereby facilitates design space exploration across a wide range of these trade-offs. For example, vectorization involves trade-offs involving the potential for improved throughput and exploitation of data parallelism at the expense of increased buffer memory requirements [45, 52].

36.6.4 Graph Scheduling and Mapping

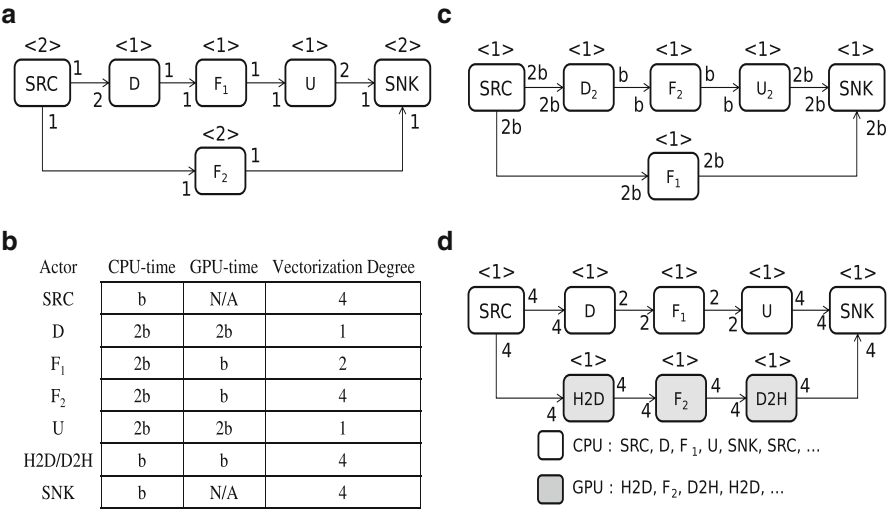
After the GLV transformation is applied to the intermediate SDF graph representation in DIF, DIF-GPU generates a schedule for the vectorized, single-rate SDF graph G_{vect}. The schedule can either be generated from a user-specified mapping configuration (assignment of actors to specific GPU and CPU resources) or computed using a selected scheduling algorithm that is implemented in the DIF package. When the user specifies the mapping configurations, DIF-GPU generates a schedule by firing the actors on each processor according to their topological order in G_{vect}.

When the user does not specify the mapping configuration, the user can select a scheduling algorithm to automatically generate the mapping and schedule. DIF-GPU integrates multiple scheduling algorithms, including a First-Come First-Serve (FCFS) scheduler and Mixed Integer Linear Programming (MILP) [52] scheduler. Providing multiple schedulers, automated code synthesis capability, and the ability to easily extend the tool with new schedulers allows the user to experiment with trade-offs associated with different scheduling techniques and select the strategy that is most appropriate in relation to the complexity of the input graph and the given design constraints.

DIF-GPU avoids redundant data transfer between CPUs and GPUs by complementary design of alternative FIFO implementations in LIDE-CUDA and usage of specialized actors for managing data transfer. In particular, DIF-GPU incorporates special data-transfer actors that are designed for optimized, model-based interprocessor communication between actors across separate memory subsystems. These data-transfer actors are called the *H2D* and *D2H* actors (recall that these abbreviations stand for host-to-device and device-to-host). *H2D* copies data from a buffer allocated on the CPU (i.e., the host) memory to the GPU (i.e., the device) memory; and conversely, *D2H* copies data from a GPU buffer to the host memory. After the scheduling process in DIF-GPU is complete, *H2D* or *D2H* actors are automatically inserted in the DIF representation for application data-flow graph edges that involve communication between host and device memory. This insertion of data-transfer actors is performed as an automated post-processing step both for user-specified and automatically generated mappings.

For example, in Fig. 36.5d, F_2 is mapped onto a GPU, so *H2D* is inserted between *src* and F_2, and *D2H* is inserted between F_2 and *snk*. This method employed by DIF-GPU to handle data transfer between processors aims to free the LIDE-CUDA actor designer from having to implement details of interprocessor communication and synchronization and to reduce data transfer overhead.

As a simple example to concretely demonstrate the DIF-GPU workflow, Fig. 36.5a and b show an SDF graph with execution time estimates that are proportional to the VF b. Such execution time profiles can be provided through actor-level benchmarking and then used as input to the scheduling phase in DIF-GPU. The target platform in this example is assumed to consist of a single CPU and single GPU. Brackets above the actors indicate the repetitions vector components



Actor	CPU-time	GPU-time	Vectorization Degree
SRC	b	N/A	4
D	2b	2b	1
F_1	2b	b	2
F_2	2b	b	4
U	2b	2b	1
H2D/D2H	b	b	4
SNK	b	N/A	4

Fig. 36.5 An illustration of the DIF-GPU workflow using a simple SDF graph example. (**a**) Original SDF graph. (**b**) VF-dependent execution times on CPU and GPU. (**c**) Vectorized graph with $VF = b$. (**d**) Vectorized graph for $b = 2$ with data-transfer actors inserted, and the corresponding schedule for CPU-GPU implementation

associated with the actors. Figure 36.5c shows the vectorized graph G_{vect} when $VF = b$. Figure 36.5d shows the DIF representation that results from further transformation through the insertion of *H2D* and *D2H* actors when $b = 2$ and when F_2 is mapped onto the GPU and other actors are mapped onto the CPU.

36.6.5 Code Generation

DIF-GPU generates well-structured, human-readable CUDA source code that can be linked with LIDE-CUDA libraries and compiled with standard CUDA development tools for implementation on CPU-GPU platforms.

Figures 36.6 and 36.7 show the generated LIDE-CUDA header and implementation file code for the sample graph in Fig. 36.5d. The generated code consists mainly of the *constructor*, *execute function*, and *destructor* for the synthesized SDF graph implementation. The constructor instantiates all of the actors and edges in the data-flow graph and connects the actors and edges according to the graph topology. The edges are assigned capacities, token sizes, and memory spaces automatically based on information in the DIF language graph specification, and on graph analysis techniques in the DIF package. The actors are assigned to processors based on the user-specified or auto-generated mapping information.

The generated code also initializes data structures for the LIDE-CUDA multi-thread scheduler. The execute function for the synthesized SDF graph

```
/* Include headers */
/* ... */

#define SNK 5
#define H2D_0 6
#define F1 4
#define D2 1
#define F2 2
#define U2 3
#define D2H_0 7
#define SRC 0
#define ACTOR_COUNT 8
#define NUMBER_OF_THREADS 2

class sample_graph {
public:
    sample_graph();
    ~sample_graph();
    void execute();
private:
    lide_cuda_thread_list* thread_list;
    lide_c_actor_context_type* actors[ACTOR_COUNT];
    char *descriptors[ACTOR_COUNT];
    lide_cuda_fifo_pointer edge_in_h2d_0;
    lide_cuda_fifo_pointer e1;
    /* Other FIFO declarations */
    /* ... */
};
```

Fig. 36.6 Generated header file for the example SDF graph of Fig. 36.5d

implementation starts the multi-thread scheduler and creates the threads. The threads then proceed to execute actor firings based on the mapping decisions embodied in the generated code. The destructor terminates the threads and actor structures and releases allocated memory.

36.6.6 Testing in DIF-GPU Using DICE

DIF-GPU employs DICE for unit testing in all parts of the workflow. The DIF-GPU framework is developed using a combination of Java, C, and CUDA; therefore, the multi-language support in DICE is useful for testing of the all components within the DIF-GPU framework. Components in DIF-GPU that require unit testing include (1) relevant data-flow transformation and scheduling techniques that apply

```
#include "sample_graph.h"

/* Full constructor */
sample_graph::sample_graph(){
    /* Create edges */
    edge_in_h2d_0 = lide_cuda_fifo_new(4, sizeof(float), CPU);
    e1 = lide_cuda_fifo_new(4, sizeof(float), CPU);
    e3 = lide_cuda_fifo_new(2, sizeof(float), CPU);
    edge_out_d2h_0 = lide_cuda_fifo_new(4, sizeof(float), CPU);
    edge_out_h2d_0 = lide_cuda_fifo_new(4, sizeof(float), GPU);
    edge_in_d2h_0 = lide_cuda_fifo_new(4, sizeof(float), GPU);
    e2 = lide_cuda_fifo_new(2, sizeof(float), CPU);
    e4 = lide_cuda_fifo_new(4, sizeof(float), CPU);
    /* Create actors */
    actors[SRC] = (lide_c_actor_context_type*)
            lide_cuda_src2_new(e1,edge_in_h2d_0,4,4, CPU);
    actors[SNK] = (lide_c_actor_context_type*)
            lide_cuda_snk2_new(e4,edge_out_d2h_0,4,4, CPU);
    actors[H2D_0] = (lide_c_actor_context_type*)
            lide_cuda_memcpy_new(edge_in_h2d_0,edge_out_h2d_0,4,4,
            sizeof(float), GPU);
    actors[F1] = (lide_c_actor_context_type*)
            lide_cuda_f_new(edge_out_h2d_0,edge_in_d2h_0,4,4, GPU);
    actors[D2] = (lide_c_actor_context_type*)
            lide_cuda_d_new(e1,e2,4,2, CPU);
    actors[F2] = (lide_c_actor_context_type*)
            lide_cuda_f_new(e2,e3,2,2, CPU);
    actors[U2] = (lide_c_actor_context_type*)
            lide_cuda_u_new(e3,e4,2,4, CPU);
    actors[D2H_0] = (lide_c_actor_context_type*)
            lide_cuda_memcpy_new(edge_in_d2h_0,edge_out_d2h_0,4,4,
            sizeof(float), GPU);
    /* Create thread list */
    thread_list = lide_cuda_thread_list_init(NUMBER_OF_THREADS,
            actors, ACTOR_COUNT);
}
sample_graph::~sample_graph(){
    lide_cuda_thread_list_terminate(thread_list);
    lide_cuda_fifo_free(edge_in_h2d_0);
    /* free other fifos */
    /* ... */
    lide_cuda_src2_terminate((lide_cuda_src2_context_type*)actors[SRC]);
    /* terminate other actors */
    /* ... */
}
void sample_graph::execute(){
    /* Start thread list */
    lide_cuda_thread_list_scheduler(thread_list);
}
```

Fig. 36.7 Generated source code file for the example SDF graph of Fig. 36.5d

Table 36.2 Summary of standard files employed in ITSs for DICE-based testing in DIF-GPU

File name	DIF	LIDE-CUDA
`dlcconfig`	N/A	Specifies header and library paths
`dljconfig`	Specifies class paths	N/A
`makeme`	Invoke `javac` with settings specified in `dljconfig`	Invoke `nvcc` compiler with settings specified in `dlcconfig`
`runme`	Run test on Java VM	Run compiled test executable
`correct-output.txt`	Standard output if test executes as expected	
`expected-errors.txt`	Standard error output if test executes as expected	

the (Java-based) DIF package; (2) FIFO and actor implementations for application graph components; and (3) synthesized software for the targeted CPU-GPU implementation.

By applying the language-agnostic testing features of DICE described in Sect. 36.5, DIF-GPU provides a unified approach to implementing and managing tests for different components in DIF-GPU, as well as DSP applications and subsystems that are developed using DIF-GPU. A summary of standard files that are employed in the implementation of DICE-based tests in DIF-GPU is listed in Table 36.2.

To automatically test components in the DIF-GPU framework, we use the DICE `dxtest` utility. This utility recursively traverses all ITSs (individual test subdirectories) in the given test suite. For each ITS, `dxtest` first executes `makeme` to perform any compilation needed for the test, followed by `runme` to exercise the test. The `dlcconfig` and `dljconfig` scripts listed in Table 36.2 specify compiler configurations that are employed by the corresponding `makeme` scripts. For each ITS, `dxtest` compares the standard output generated by `runme` with `correct-output.txt` and the actual standard error output with `expected-errors.txt`. Finally, `dxtest` produces a summary of successful and failed tests, including the specific directory paths of any failed tests. In this way, the test-execution process is largely automated and simplified while operating within an integrated environment across the different Java, C, and CUDA components that need to be tested.

36.7 Summary

This chapter has covered the DSPCAD Framework, which provides an integrated set of tools for model-based design, implementation, and testing of signal processing systems. The DSPCAD Framework addresses challenges in Hardware/Software Codesign (HSCD) for signal processing involving the increasing diversity in

relevant data-flow modeling techniques, actor implementation languages, and target platforms. Our discussion of the DSPCAD Framework has focused on its three main component tools – the Data-flow Interchange Format (DIF), Lightweight Data-flow Environment (LIDE), and DSPCAD Integrative Command Line Environment (DICE) – which support flexible design experimentation and orthogonalization across abstract data-flow models, actor implementation languages, and integration with platform-specific design tools, respectively. Active areas of ongoing development in the DSPCAD Framework include data-flow techniques and libraries for networked mobile platforms, multi-core processors, and graphics processing units, as well as efficient integration with multimodal sensing platforms.

Acknowledgments Research on the DSPCAD Framework has been supported in part by the US National Science Foundation, Laboratory for Telecommunication Sciences, and Tekes – The Finnish Funding Agency For Innovation.

References

1. Beck K et al (2015) Manifesto for agile software development (2015). http://www.agilemanifesto.org/. Visited on 26 Dec 2015
2. Bhattacharya B, Bhattacharyya SS (2000) Parameterized dataflow modeling of DSP systems. In: Proceedings of the international conference on acoustics, speech, and signal processing, Istanbul, pp 1948–1951
3. Bhattacharyya SS, Deprettere E, Leupers R, Takala J (eds) (2013) Handbook of signal processing systems, 2nd edn. Springer, New York. ISBN:978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online)
4. Bhattacharyya SS, Plishker W, Shen C, Gupta A (2011) Teaching cross-platform design and testing methods for embedded systems using DICE. In: Proceedings of the workshop on embedded systems education, Taipei, pp 38–45
5. Bhattacharyya SS, Plishker W, Shen C, Sane N, Zaki G (2011) The DSPCAD integrative command line environment: introduction to DICE version 1.1. Technical report UMIACS-TR-2011-10, Institute for Advanced Computer Studies, University of Maryland at College Park. http://drum.lib.umd.edu/handle/1903/11422
6. Bilsen G, Engels M, Lauwereins R, Peperstraete JA (1996) Cyclo-static dataflow. IEEE Trans Signal Process 44(2):397–408
7. Buck JT, Lee EA (1993) Scheduling dynamic dataflow graphs using the token flow model. In: Proceedings of the international conference on acoustics, speech, and signal processing, Minneapolis
8. Choi J, Oh H, Kim S, Ha S (2012) Executing synchronous dataflow graphs on a SPM-based multicore architecture. In: Proceedings of the design automation conference, San Francisco, pp 664–671
9. Desnos K, Pelcat M, Nezan J, Bhattacharyya SS, Aridhi S (2013) PiMM: parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration. In: Proceedings of the international conference on embedded computer systems: architectures, modeling, and simulation, Samos, pp 41–48
10. Eker J, Janneck JW (2003) CAL language report, language version 1.0 – document edition 1. Technical report UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley
11. Eker J, Janneck JW (2012) Dataflow programming in CAL – balancing expressiveness, analyzability, and implementability. In: Proceedings of the IEEE Asilomar conference on signals, systems, and computers, Pacific Grove, pp 1120–1124

12. Falk J, Keinert J, Haubelt C, Teich J, Bhattacharyya SS (2008) A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In: Proceedings of the international conference on embedded software, Atlanta, pp 189–198
13. Gagnon E (1998) SableCC, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal
14. Ghamarian AH, Stuijk S, Basten T, Geilen MCW, Theelen BD (2007) Latency minimization for synchronous data flow graphs. In: Proceedings of the Euromicro conference on digital system design architectures, methods and tools, pp 189–196
15. Gregg C, Hazelwood K (2011) Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In: Proceedings of the IEEE international symposium on performance analysis of systems and software, Austin, pp 134–144
16. Gu R, Janneck J, Raulet M, Bhattacharyya SS (2009) Exploiting statically schedulable regions in dataflow programs. In: Proceedings of the international conference on acoustics, speech, and signal processing, Taipei, pp 565–568
17. Gu R, Piat J, Raulet M, Janneck JW, Bhattacharyya SS (2010) Automated generation of an efficient MPEG-4 reconfigurable video coding decoder implementation. In: Proceedings of the conference on design and architectures for signal and image processing, Edinburgh
18. Hamill P (2004) Unit test frameworks. O'Reilly & Associates, Inc., Sebastopol
19. Haubelt C, Falk J, Keinert J, Schlichter T, Streubühr M, Deyhle A, Hadert A, Teich J (2007) A SystemC-based design methodology for digital signal processing systems. EURASIP J Embed Syst 2007: 22. Article ID 47580
20. Hierons RM et al (2009) Using formal specifications to support testing. ACM Comput Surv 41(2):1–22
21. Hsu C, Corretjer I, Ko M, Plishker W, Bhattacharyya SS (2007) Dataflow interchange format: language reference for DIF language version 1.0, user's guide for DIF package version 1.0. Technical report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park. Also Computer Science Technical Report CS-TR-4871
22. Hsu C, Keceli F, Ko M, Shahparnia S, Bhattacharyya SS (2004) DIF: an interchange format for dataflow-based design tools. In: Proceedings of the international workshop on systems, architectures, modeling, and simulation, Samos, pp 423–432
23. Hsu C, Ko M, Bhattacharyya SS (2005) Software synthesis from the dataflow interchange format. In: Proceedings of the international workshop on software and compilers for embedded systems, Dallas, pp 37–49
24. Hunt A, Thomas D (2003) Pragmatic unit testing in Java with JUnit. The Pragmatic Programmers
25. Janneck JW, Mattavelli M, Raulet M, Wipliez M (2010) Reconfigurable video coding: a stream programming approach to the specification of new video coding standards. In: Proceedings of the ACM SIGMM conference on multimedia systems, New York, pp 223–234
26. Kedilaya S, Plishker W, Purkovic A, Johnson B, Bhattacharyya SS (2011) Model-based precision analysis and optimization for digital signal processors. In: Proceedings of the European signal processing conference, Barcelona, pp 506–510
27. Keinert J, Haubelt C, Teich J (2006) Modeling and analysis of windowed synchronous algorithms. In: Proceedings of the international conference on acoustics, speech, and signal processing, Toulous
28. Keutzer K, Malik S, Newton R, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput Aided Des Integr Circuits Syst 19:1523-1543
29. Kwok Y (1997) High-performance algorithms for compile-time scheduling of parallel processors. Ph.D. thesis, The Hong Kong University of Science and Technology
30. Lee EA, Messerschmitt DG (1987) Synchronous dataflow. Proc IEEE 75(9):1235–1245
31. Lee EA, Parks TM (1995) Dataflow process networks. Proc IEEE 83:773–799
32. Lin S, Liu Y, Plishker W, Bhattacharyya SS (2016) A design framework for mapping vectorized synchronous dataflow graphs onto CPU–GPU platforms. In: Proceedings of the international workshop on software and compilers for embedded systems, Sankt Goar, pp 20–29

33. Lin S, Wang LH, Vosoughi A, Cavallaro JR, Juntti M, Boutellier J, Silvén O, Valkama M, Bhattacharyya SS (2015) Parameterized sets of dataflow modes and their application to implementation of cognitive radio systems. J Signal Process Syst 80(1):3–18
34. Murthy PK, Lee EA (2002) Multidimensional synchronous dataflow. IEEE Trans Signal Process 50(8):2064–2079
35. Neuendorffer S, Lee E (2004) Hierarchical reconfiguration of dataflow models. In: Proceedings of the international conference on formal methods and models for codesign, San Diego
36. Oh H, Dutt N, Ha S (2006) Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In: Proceedings of the Asia South Pacific design automation conference, Yokohama, pp 497–502
37. Pelcat M, Aridhi S, Piat J, Nezan JF (2013) Physical layer multi-core prototyping. Springer, London
38. Pelcat M, Desnos K, Heulot J, Nezan JF, Aridhi S (2014) Dataflow-based rapid prototyping for multicore DSP systems. Technical report PREESM/2014-05TR01, Institut National des Sciences Appliquées de Rennes
39. Pelcat M, Desnos K, Maggiani L, Liu Y, Heulot J, Nezan JF, Bhattacharyya SS (2015) Models of architecture. Technical report PREESM/2015-12TR01, IETR/INSA Rennes. HAL Id: hal-01244470
40. Pelcat M, Menuet P, Aridhi S, Nezan JF (2009) Scalable compile-time scheduler for multi-core architectures. In: Proceedings of the design, automation and test in Europe conference and exhibition, Nice, pp 1552–1555
41. Pelcat M, Piat J, Wipliez M, Aridhi S, Nezan JF (2009) An open framework for rapid prototyping of signal processing applications. EURASIP J Embed Syst 2009:Article No. 11
42. Plishker W, Sane N, Bhattacharyya SS (2009) A generalized scheduling approach for dynamic dataflow applications. In: Proceedings of the design, automation and test in Europe conference and exhibition, Nice, pp 111–116
43. Plishker W, Sane N, Kiemb M, Anand K, Bhattacharyya SS (2008) Functional DIF for rapid prototyping. In: Proceedings of the international symposium on rapid system prototyping, Monterey, pp 17–23
44. Plishker W, Sane N, Kiemb M, Bhattacharyya SS (2008) Heterogeneous design in functional DIF. In: Proceedings of the international workshop on systems, architectures, modeling, and simulation, Samos, pp 157–166
45. Ritz S, Pankert M, Meyr H (1993) Optimum vectorization of scalable synchronous dataflow graphs. In: Proceedings of the international conference on application specific array processors, Venice
46. Sane N, Kee H, Seetharaman G, Bhattacharyya SS (2011) Topological patterns for scalable representation and analysis of dataflow graphs. J Signal Process Syst 65(2):229–244
47. Shen C, Plishker W, Wu H, Bhattacharyya SS (2010) A lightweight dataflow approach for design and implementation of SDR systems. In: Proceedings of the wireless innovation conference and product exposition, Washington, DC, pp 640–645
48. Shen C, Wang L, Cho I, Kim S, Won S, Plishker W, Bhattacharyya SS (2011) The DSPCAD lightweight dataflow environment: introduction to LIDE version 0.1. Technical report UMIACS-TR-2011-17, Institute for Advanced Computer Studies, University of Maryland at College Park. http://hdl.handle.net/1903/12147
49. Sriram S, Bhattacharyya SS (2009) Embedded multiprocessors: scheduling and synchronization, 2nd edn. CRC Press, Boca Rato. ISBN:1420048015
50. T Dohmke HG (2007) HG test-driven development of a PID controller. IEEE Soft 24(3):44–50
51. Theelen BD, Geilen MCW, Basten T, Voeten JPM, Gheorghita SV, Stuijk S (2006) A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of the international conference on formal methods and models for codesign, Napa
52. Zaki G, Plishker W, Bhattacharyya SS, Clancy C, Kuykendall J (2013) Integration of dataflow-based heterogeneous multiprocessor scheduling techniques in GNU radio. J Signal Process Syst 70(2):177–191. doi:10.1007/s11265-012-0696-0

Control/Architecture Codesign for Cyber-Physical Systems

37

Wanli Chang, Licong Zhang, Debayan Roy, and Samarjit Chakraborty

Abstract

Control/architecture codesign has recently emerged as one popular research focus in the context of cyber-physical systems. Many of the cyber-physical systems pertaining to industrial applications are embedded control systems. With the increasing size and complexity of such systems, the resource awareness in the system design is becoming an important issue. Control/architecture codesign methods integrate the design of controllers and the design of embedded platforms to exploit the characteristics on both sides. This reduces the design conservativeness of the separate design paradigm while guaranteeing the correctness of the system and thus helps to achieve more efficient design. In this chapter of the handbook, we provide an overview on the control/architecture codesign in terms of resource awareness and show three illustrative examples of state-of-the-art approaches, targeting respectively at communication-aware, memory-aware, and computation-aware design.

Acronyms

CFG	Control-Flow Graph
CPS	Cyber-Physical System
DSE	Design Space Exploration
ECU	Electronic Control Unit
E/E	Electric and Electronic
EMB	Electro-Mechanical Brake
ET	Event-Triggered
FTDMA	Flexible Time Division Multiple Access

W. Chang (✉)
Singapore Institute of Technology, Singapore, Singapore
e-mail: wanli.chang@singaporetech.edu.sg

L. Zhang • D. Roy • S. Chakraborty
TU Munich, Munich, Germany
e-mail: licong.zhang@tum.de; debayan.roy@tum.de; samarjit@tum.de

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_37

LCS	Live Cache States
MILP	Mixed Integer Linear Programming
OS	Operating System
PSO	Particle Swarm Optimization
RCS	Reaching Cache States
RTOS	Real-Time Operating System
TDMA	Time-Division Multiple Access
TT	Time-Triggered
WCET	Worst-Case Execution Time

Contents

37.1 Introduction

Cyber-physical systems refer to systems where tight interaction between the computational elements (cyber) and the physical entities (physical) is emphasized. A typical example of a cyber-physical system is an embedded control system. In such a system, software implementation of the controllers running on processing units are used to control physical plants. As shown in Fig. 37.1, the processing units are connected with sensors and actuators where the sensors measure the states of the plants, the controllers compute the control input, and the actuators apply the control input onto the physical plants. Today, cyber-physical systems have become commonplace and can be found in the domains like automotive, avionics, industrial automation, chemical engineering, etc. The automotive Electric and Electronic (E/E) system is an example of such a system. In a modern vehicle, increasingly more functions are realized by software mapped on the Electronic Control Unit (ECU). These include

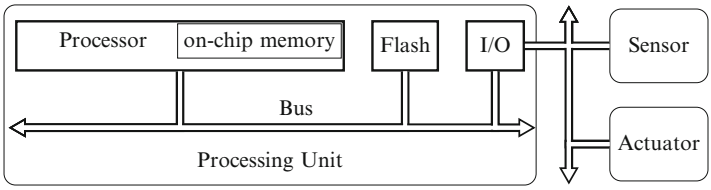


Fig. 37.1 A processing unit with a processor and on-chip memory for program execution. Instructions are stored in the flash memory. Programmable I/O peripherals are used for communication with sensors, actuators, and other processing units. For instance, Infineon XC23xxB Series, which is widely used in automotive systems, has a single processor with a minimum operating frequency of 20MHz. It is typically equipped with a small size of on-chip SRAM memory and up to 256 kB flash memory [7].

the functions for vehicle dynamics control, body components control (e.g., doors and lights), infotainment, and advanced driver assistance systems (ADAS). Some of these functions have stringent timing requirements, and some demand processing and transport of intensive data amount. The characteristics and performance of the cyber part, i.e., the electronics and software, strongly influence the performance of the physical part. In the case of safety-critical control functions, the timing properties of the software implementation of the controllers, e.g., the sampling period and the sensor-to-actuator delay, play a vital role in the control performance. Therefore, with the Cyber-Physical System (CPS)-oriented thinking, more attention is necessary for the implementation of the controllers in an embedded platform and interplay between the embedded platform design and the control design.

The hardware architecture of the computational part of a cyber-physical system consists mainly of one or more processing units. In case of a multiprocessor architecture, the processing units are commonly connected by a communication network, where data between different processing units can be transmitted. Typical communication networks in this context include the FlexRay [3], CAN [13], LIN [4], and MOST [5] in the automotive domain; AFDX ţteAFDX and AS6802 [6] in the avionics domain; and Profibus, Profinet [33], and EtherCAT in the industrial automation domain. ▸ Chapter. 24, "Networked Real-Time Embedded Systems" provides a more detailed study on some important real-time communication protocols. These communication protocols implement different data transmission approaches and are each suitable for a specific set of requirements. On each processing unit, the computation is performed by tasks, each of which is typically implemented by a piece of code. Multiple tasks can be grouped together to form an application, where an independent function (e.g., a feedback control loop) is performed. In a distributed application, where the tasks are mapped onto different processing units, the data between the relevant tasks are transmitted over the communication network. It is common that multiple tasks belonging to the same or different applications are mapped on one processing unit. In this case, an operating system (e.g., OSEK [1], eCos [18]) is sometimes used to coordinate task executions and allocate resources for the tasks.

In many embedded systems in the context of cyber-physical systems, the applications are control applications, where the software implementation of the controllers controls physical plants [28]. The design of controllers for these applications from a control-theoretical perspective are well established. The control design methods can be drawn from a large pool of research and practical expertise and experience that have been accumulated in the control community in the past few decades. However, little attention has been paid to the actual implementation of the controllers in the embedded platforms. In this case, not only the control theoretical aspect of the design problem needs to be taken into account, e.g., type of controllers and control gains, but also the characteristics of the underlying embedded platforms. The design aspects on the embedded system side include, for example, the task partitioning and mapping, the scheduling of tasks and communication, and the allocation of memory and cache. There is a tight interconnection between the control and the embedded platform design [16]. For example, the results of the embedded platform design can strongly influence the control performance through properties like sampling period, delay, and jitter. Reversely, the requirements from the control design side also influence the platform design. Conventionally the control and embedded platform design are done separately and then integrated afterward. In this case, the engineers on both sides need to make assumptions of the other side. Since most control applications are safety critical, such assumptions are inevitably quite conservative to guarantee the safety of the control applications. Due to this conservativeness, usually the resources on the embedded platform, e.g., computation, communication, memory, and energy resources, are not optimally utilized. On the other hand, these resources on an embedded platform are quite limited, constrained by the size and cost reasons. In recent years, both the size and complexity of the embedded systems in industrial domains have increased drastically. In the automotive domain, for example, a modern premium passenger car can contain up to 100 million lines of software code [17]. In such a computation and data-intensive platform, resource-efficient design has become a quite important issue. Therefore, the CPS community has become increasingly conscious that some systematic design methods will be necessary for design of resource-aware embedded control systems.

The resources on an embedded platform can be divided into different categories, e.g., computation, communication, memory, energy, and input/output interfaces. In the context of this chapter, three of the most important resources, namely, the computation resource, the communication resource, and the memory resource, are considered. In the following paragraphs, each of the aforementioned resources will be explained in detail.

Communication resources can generally be represented as the bandwidth of a communication bus or a network link, which denotes the number of bits that can be transmitted per second. Therefore, there is only a limited amount of data that can be transmitted within a specific time frame. More precise characterization of the communication resource, however, is protocol specific. The communication protocols implement different data transmission approaches, which can be broadly divided into two different categories, namely, the Time-Triggered (TT) paradigm and the Event-Triggered (ET) paradigm. For example, a Time-Division Multiple Access

(TDMA) bus is a typical time-triggered bus communication. In this case, a period of time is divided into multiple time slots, and the usage of the communication resource can be represented directly by the number of utilized slots. In an industrial-sized distributed embedded system, the communication resource is quite constrained. As the size of the system increases, more processing units and data can be incrementally mapped on to the communication bus or network. However, the bandwidth of a communication protocol cannot be easily increased. Therefore, communication-efficient design could enable the system to accommodate more applications or enhance the performance of the applications. Related to this, in recent years, there have been several works on integrated controller synthesis and task and message scheduling of distributed embedded control systems, e.g., [20, 23, 34, 35]. However, most of these works, e.g., [20, 23, 34] only consider optimization of control performance while satisfying communication constraints. In addition, there have been several works, e.g., [29, 39] on schedule optimization of distributed time-triggered embedded systems where the objective is to minimize communication bandwidth utilization while satisfying timing constraints. However, these works do not consider control applications.

Memory resources mainly refer to the size of cache due to its high cost. Within a processing unit, there are typically two levels of memory – cache and main memory. In Fig. 37.1, the on-chip memory works as cache and the flash memory serves as the main memory. The main memory has a large size and can thus store all the application programs and data, but experiences high read/write latencies (hundreds of processor cycles). The cache is faster (several processor cycles), but usually limited in size. In this chapter, the focus is on instruction memory. It is assumed that the access times of cache and main memory are t_c and t_m, respectively, where $t_c \ll t_m$. When a processor executes an instruction, it checks the cache first. If this instruction is located in the cache, it is a cache hit and the access time is t_c. If this instruction is not in the cache, the memory block containing it is fetched from the main memory and then written into cache. This is called a cache miss and the access time is t_m. Afterward, when the same instruction is called again by the processor, the access time is t_c if it is still in the cache without being replaced. Increasing the cache size and improving the cache reuse are two general methods to reduce the execution time of a program. A program usually has different execution paths resulting in different execution times, depending on the input. The Worst-Case Execution Time (WCET) is defined to be the maximum length of time a program takes to be executed. The WCET constrains the sampling period of a control application, which is defined to be the duration between two consecutive executions of a control program, and thus has significant impact on the control performance. In resource-aware embedded control systems design, it is desirable to minimize the cache size while satisfying the performance requirement or, equivalently, improve the performance for a given memory. Therefore, on one hand, the cache reuse should be maximized, and on the other hand, the controller must be suitably designed to exploit the shortened sampling periods. There have been some works on cache reuse maximization by employing code positioning during compile time [22, 26, 32] and also during run time [11], but these cannot directly be applied to embedded

control systems as code rearrangement would impact the timing properties, and this is difficult to incorporate while designing the controllers.

Computation resources usually mean the available execution time of a processor, when the processing speed is given. Considering multiple applications sharing one single-core processing unit, each application is allocated a certain period of execution time. In general, the performance of an application can be improved if it is allowed to access the processor longer. On a processor sometimes runs an Operating System (OS). For instance, ERCOSek [1, 19] is a widely used time-triggered OS on ECUs and only offers a limited set of predefined periods. It implies that the sampling periods of control applications have to be taken from this set. Generally, a shorter sampling period allows the controller to respond to its plant more frequently and is thus potentially able to achieve better control performance with an appropriately designed controller. The obvious downside is a higher processor utilization, which is defined to be the WCET of an application divided by its sampling period. This prevents more functions and applications from being integrated onto the processing unit. Therefore, the controller should use the largest possible sampling period that is able to fulfill the control performance requirement and satisfy the system constraints. In most cases, the optimal sampling period is not directly realizable on the OS. The conventional way to handle it is to use the largest sampling period offered by the OS that is smaller than the optimal one. This is a straightforward method, but leads to a waste of computational resources. Toward this, there have been several works on state-feedback-based optimal resource allocation to the control loops sharing the same processor, e.g., [14, 15, 21, 25, 30, 31]. All these works focus on online assignment of sampling periods of the control loops based on the system dynamics like plant states, disturbance, or error. However, an online decision-making must be very fast to be effective, and therefore, there must be some heuristics involved. Therefore, an offline schedule computation that guarantees performance and reduces the processor utilization will be more desirable.

The rest of this chapter is organized as follows. In Sect. 37.2, the basics of feedback control applications are briefly reviewed. In the three sections that follow, three state-of-art approaches of different aspects in terms of resource-aware algorithm/architecture codesign are explained, namely, the communication-aware design (Sect. 37.3), the memory-aware design (Sect. 37.4), and the computation-aware design (Sect. 37.5). Finally, Sect. 37.6 contains the concluding remarks.

37.2 Embedded Control Systems

In this section, some background knowledge for the embedded control systems considered in this chapter is provided. Firstly, a brief introduction in the embedded systems architecture is provided. Then the basics of feedback control systems as well as the control performance metrics and the method for optimal pole placement are explained.

37.2.1 Embedded Systems Architecture

The architecture considered in this chapter does not refer to the processor architecture, but the design parameters for the underlying hardware and the communication for the embedded controllers. The architecture can either be a single ECU, as shown in Fig. 37.1, or a distributed system consisting of multiple ECUs connected by a communication network, as shown in Fig. 37.2. An embedded controller mapped on such an architecture is usually implemented with one or multiple tasks, where each task is a piece of software code running on the processor. A controller can be partitioned into the sensor task, the controller task, and the actuator task. The sensor task measures the state of the physical plant, the controller task computes the control input, and the actuator task applies the control input onto the physical plant. In a single processor architecture, these tasks are executed on the same ECU, while in a distributed architecture, the sensor, controller, and actuator tasks can also be mapped on different ECUs and the data between the tasks are transferred over the network as messages. It is also common that tasks of different controllers are mapped on common ECUs, where the communication, computation, and memory resources are shared between these control applications. Therefore, how to allocate the resources for the software implementation of the controllers forms the problem of architecture design. More specifically, these design parameters may include the task partition and mapping, the task and network scheduling, the use of the cache, etc. Towards the design of these parameters, ▸ Chap. 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration" provides an overview of successful approaches for system-level design space exploration for complex embedded systems.

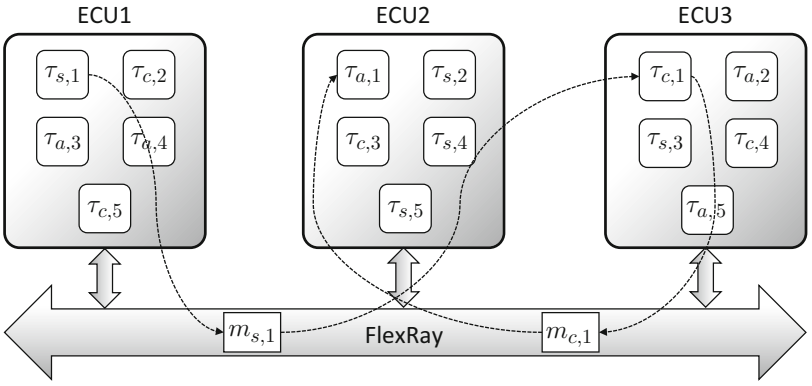


Fig. 37.2 An example of a distributed architecture for the embedded control systems. This example consists of 3 ECUs connected by a FlexRay bus. Five control applications are mapped on this architecture, where $\tau_{s,i}$, $\tau_{c,i}$, and $\tau_{a,i}$ denote respectively the sensor task, controller task, and actuator task of the ith control application. Two messages over the communication bus for first control application as well as the data dependency are shown

37.2.2 Feedback Control Systems

Throughout this chapter, linear single-input single-output (SISO) control applications are considered. The dynamic behavior is modeled by a set of differential equations,

$$\dot{x}(t) = Ax(t) + Bu(t), \quad y(t) = Cx(t), \tag{37.1}$$

where $x(t) \in \mathbb{R}^n$ is the system state, $y(t)$ is the system output, and $u(t)$ is the control input. The number of system states is n. A, B, and C are system matrices of appropriate dimensions. System poles are eigenvalues of A. In a state-feedback control algorithm, $u(t)$ is computed utilizing $x(t)$ (feedback signals) and is then applied to the plant, which is expected to achieve certain desired behavior. In an embedded implementation platform, the operations (measure $x(t)$, compute $u(t)$, and apply $u(t)$) of a control loop are performed only at discrete time instants. In the case where the sensor-to-actuator delay d is ignored, the continuous-time system in (37.1) can be transformed into a discrete-time system with the sampling period h which can be represented as [10]

$$x[k+1] = A_d x[k] + B_d u[k], \quad y[k] = C_d x[k], \tag{37.2}$$

where sampling instants are $t = t_k$ $(k = 1, 2, 3, \cdots)$ and $h = t_{k+1} - t_k$. $x[k]$ and $u[k]$ are the values of $x(t)$ and $u(t)$ at $t = t_k$ and

$$A_d = e^{Ah}, \; B_d = \int_0^h (e^{At} dt) \cdot B, \; C_d = C. \tag{37.3}$$

A system is asymptotically stable if the steady-state impulse response is zero, i.e., $\lim_{k\to\infty} y_\delta[k] = 0$. Toward this, $u[k]$ needs to be designed utilizing the states $x[k]$ in a state-feedback controller. The general representation is as follows:

$$u[k] = K_d \cdot x[k] + F_d \cdot r, \tag{37.4}$$

where K_d is the feedback gain, F_d is the feedforward gain, and r is the reference value. Then, the system dynamics in (37.2) becomes

$$x[k+1] = (A_d + B_d K_d)x[k] + B_d F_d r, \tag{37.5}$$

i.e., closed-loop dynamics. Different locations of closed-loop system poles, i.e., eigenvalues of $(A_d + B_d K_d)$, result in different system behaviors. Pole locations can be decided by the pole-placement technique, and then the following characteristics equation of H can be constructed with these poles as roots:

$$H^n + \gamma_1 H^{n-1} + \gamma_2 H^{n-2} + \cdots + \gamma_n = 0. \tag{37.6}$$

Define

$$\gamma_c(A_d) = A_d^n + \gamma_1 A_d^{n-1} + \gamma_2 A_d^{n-2} + \cdots + \gamma_n \mathbf{I}, \tag{37.7}$$

where $\mathbf{I}$ is the n-dimensional identity matrix. According to Ackermann's formula [8], the feedback gain to stabilize the system is calculated as

$$K_d = -[0 \ \cdots \ 0 \ 1]\, \zeta^{-1}\, \gamma_c(A_d), \tag{37.8}$$

where ζ represents the controllability matrix of the system and is given by

$$\zeta = [B_d \ \ A_d B_d \ \ \ldots \ \ A_d^{n-1} B_d] \tag{37.9}$$

The static feedforward gain F is designed to achieve $y[k] \to r$ as $k \to \infty$ and can be computed by

$$F_d = \frac{1}{C_d(\mathbf{I} - A_d - B_d K_d)^{-1} B_d}. \tag{37.10}$$

However, in a realistic implementation of a control application, a non-negligible sensor-to-actuator delay needs to be taken into account. In the case, where the delay is smaller or equal to one sampling period, i.e., $0 \le d \le h$, the discrete-time system in (37.2) becomes a sampled-data system [12] as

$$x[k+1] = A_d x[k] + B_{d1}(d)u[k-1] + B_{d0}(d)u[k], \tag{37.11}$$

where

$$B_{d0}(D_c) = \int_0^{h-d} e^{At}dt \cdot B, \ B_{d1}(d) = \int_{h-d}^{h} e^{At}dt \cdot B. \tag{37.12}$$

In (37.11), it is assumed that $u[-1] = 0$ for $k = 0$. Notice that $x[k+1]$ depends on both $u[k]$ and $u[k-1]$, since during the sensor-to-actuator delay, $u[k]$ is not available and $u[k-1]$ is applied to the plant. A new system state $z[k] = \left[\, x[k] \ u[k-1] \,\right]^T$ is defined, and the transformed system becomes

$$z[k+1] = A_{aug} z[k] + B_{aug} u[k], \quad y[k] = C_{aug} z[k], \tag{37.13}$$

where

$$A_{aug} = \begin{bmatrix} A_d & B_{d1}(d) \\ 0 & 0 \end{bmatrix}, \ B_{aug} = \begin{bmatrix} B_{d0}(d) \\ \mathbf{I} \end{bmatrix}, \ C_{aug} = \begin{bmatrix} C_d \ 0 \end{bmatrix}. \tag{37.14}$$

Next, apply the following input signal:

$$u[k] = K_{aug} \cdot z[k] + F_{aug} \cdot r. \tag{37.15}$$

The closed-loop system is then

$$z[k+1] = (A_{aug} + B_{aug} K_{aug}) z[k] + B_{aug} F_{aug} r. \tag{37.16}$$

The feedback gain K_{aug} can then be calculated according to (37.8) by replacing A_d with A_{aug} and also replacing A_d and B_d with A_{aug} and B_{aug} while computing the controllability matrix ζ in (37.9). Similarly, the feedforward gain F_{aug} is computed according to (37.10) by replacing A_d, B_d, C_d, and K_d with A_{aug}, B_{aug}, C_{aug}, and K_{aug}, respectively.

37.2.2.1 Control Performance Metrics

There are different metrics to measure the performance of a control system. In this chapter, two common metrics to measure the control performance are considered. (i) The *steady-state performance* of a control application which can be commonly measured by a *cost function* [35], which in the discrete case can be represented as

$$J = \sum_{k=0}^{n} [\lambda u[k]^2 + (1-\lambda)\sigma[k]^2] h, \tag{37.17}$$

where λ is a weight taking the value between 0 and 1, $u[k]$ is the control input and $\sigma[k] = |r - y[k]|$ is the tracking error. (ii) The *settling time*, ξ, where ξ denotes the time necessary for the system to reach and remain within 1% of the reference value

$$J = \xi. \tag{37.18}$$

37.2.2.2 Optimal Pole Placement

For a control application, in order to design the controller which optimizes the control performance for a given sampling period, an optimization problem for the pole placement can be formulated. Decision variables are poles of the closed-loop system. Therefore, the number of dimensions in the decision space is equal to the number of states in the closed-loop system. The objective is to optimize the value of the selected control performance metric. Absolute values of all poles have to be less than unity to ensure system stability. The control input saturation needs to be respected as well.

It is challenging to solve such a constrained non-convex optimization problem with significant nonlinearity. Here, the Particle Swarm Optimization (PSO) technique, which is highly efficient and scalable [36], can be used. A group of particles are randomly initialized in the decision space with positions and velocities. They search for the optimum by iteratively updating their positions. The search is led by

two points. The first is the local best point that has been reached by a particle. Every particle has its own local best point. The second is the global best point that has been reached considering all particles. A point that respects all constraints is always better than a point that violates at least one constraint, no matter what their objective values are. When comparing two points that either respect all constraints or violate at least one constraint, the point with a better objective value is better.

The velocity of a particle is determined by the following equation:

$$\begin{aligned} V_{\text{new}} = {} & \alpha_0 V_{\text{current}} + \alpha_1 \text{rand}(0,1)(P_{\text{lbest}} - P_{\text{current}}) \\ & + \alpha_2 \text{rand}(0,1)(P_{\text{gbest}} - P_{\text{current}}), \end{aligned} \tag{37.19}$$

where V_{new} is the new velocity, V_{current} is the current velocity, P_{current} is the current position, P_{lbest} is the local best point of this particle, and P_{gbest} is the best point of all particles. rand(0, 1) is a random number with uniform distribution from the open interval (0, 1). α_0, α_1 and α_2 are parameters that can be determined empirically. The new position of this particle is

$$P_{\text{new}} = P_{\text{current}} + V_{\text{new}}. \tag{37.20}$$

The algorithm is terminated once all particles have converged or the maximum number of iterations has been reached. The time complexity of PSO is clearly polynomial.

37.3 Communication-Aware Control/Architecture Codesign

In this section, a codesign approach that synthesizes simultaneously controllers, task, and communication schedules for a FlexRay-based ECU network will be introduced. The approach consists mainly of two stages, namely, the control design stage and the cooptimization stage. This separation is necessary because the problem deals with a large design space combining the dimensions of both control and platform design. Therefore, the whole space is partitioned into smaller subspaces while considering all feasible regions in the design space by exploiting some domain-specific characteristics. In the control design stage, an optimal controller is synthesized at each possible sampling period for each control application. This is done by using the pole-placement control design method and exploring the design space for poles using heuristics. In the cooptimization stage, a bi-objective optimization problem is formulated, and a customized method is employed to generate a number of feasible design parameter sets, where each set represents a Pareto point reflecting the trade-off between the objectives of control performance and bus utilization. Here, we will first explain the problem setting and then discuss in detail the state-of-the-art control-communication codesign technique applicable to such a setting.

37.3.1 Problem Setting

Distributed implementation: Consider a distributed architecture where a set of ECUs represented by $p_i \in \mathbb{P}$ are connected through a FlexRay bus. A number of control applications denoted by $C_i \in \mathscr{C}$ are mapped on such an embedded platform. Each control application C_i can be partitioned into three dependent application tasks: (i) *sensor task*, $\tau_{s,i}$, measures the system states (using sensors) of the physical system if measurable; (ii) *controller task*, $\tau_{c,i}$, computes the controller input based on the measured system states; and (iii) *actuator task*, $\tau_{a,i}$, applies the control input (using actuators) to the physical system. Without loss of generality, assume that the three tasks are mapped on different ECUs. Then the sensor values measured by $\tau_{s,i}$ are sent on the bus through message $f_{s,i}$, and the control input calculated by $\tau_{c,i}$ is sent as message $f_{c,i}$. The time between the start of sensor task and the end of actuator task is defined as the *sensor-to-actuator delay*, denoted as d. As shown in Fig. 37.3, this delay depends on the interplay between the task and communication schedules.

ECU task model: Here, consider the case where time-triggered non-preemptive scheduling scheme is exhibited by the Real-Time Operating System (RTOS) on the ECUs. Each task of the control applications is considered to be periodic and is defined by the tuple $\tau_{x,i} = \{O_{x,i}, P_{x,i}, E_{x,i}\}$, where $O_{x,i}$, $P_{x,i}$, and $E_{x,i}$ denote respectively the offset, the period, and the WCET of the task. Here, the subscript $x \in \{s, c, a\}$ where s, c, or a respectively identifies sensor, controller, or actuator task. The subscript i identifies the control application C_i it constitutes. Thus, if $\bar{t}(\tau_{x,i}, k)$ and $\tilde{t}(\tau_{x,i}, k)$ are defined as the starting and the latest finishing time of the k^{th} ($k \in \mathbb{Z}^*$) instance of task τ_i, then

$$\bar{t}(\tau_{x,i}, k) = O_{x,i} + kP_{x,i}, \quad \tilde{t}(\tau_{x,i}, k) = O_{x,i} + kP_{x,i} + E_{x,i}. \tag{37.21}$$

A set of *communication tasks* are required besides the application tasks. The communication task on the sending ECU writes the data produced by the application tasks into the corresponding *transmit buffers* of the communication controller, and on the receiving ECU, it reads the data from the corresponding *receive buffers* and

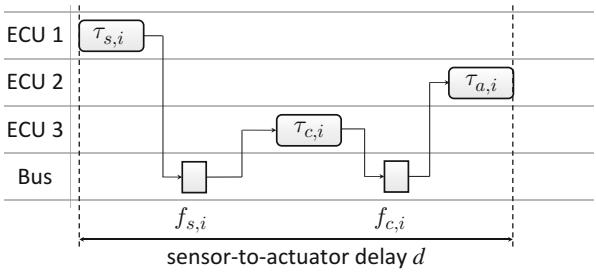


Fig. 37.3 Distributed embedded control application

forwards them to the application tasks. The nature of these communication tasks depends on the specific implementation. Here, consider that the execution time of all communication tasks is bounded by ϵ, and assume that a communication task is scheduled directly after its corresponding application task at the sending side and directly before the application task at the receiving side.

FlexRay communication: FlexRay [3] is an automotive communication protocol usually applied for safety-critical applications. Although FlexRay communication is discussed in detail in ▸ Chap. 24, "Networked Real-Time Embedded Systems", few important points are reiterated here for better understanding of the problem and the subsequent solution. Being a hybrid protocol, it offers both TT and ET communication services. FlexRay is organized as a series of *communication cycles*, the length of which is denoted as T_{bus}. Each communication cycle contains mainly the *static segment* (ST) and optionally *dynamic segment* (DYN), where the TT and ET communication services are implemented respectively. The static segment applies the TDMA scheme and is split into a number of *static slots* of equal length Δ. Here, the slots on the static segment can be represented as $\mathscr{S}_{ST} = \{1, 2, \ldots, N_s\}$, where N_s is the number of static slots. Once a static slot is assigned, if no data is sent in a specific communication cycle, the static slot will still be occupied. The dynamic segment follows a Flexible Time Division Multiple Access (FTDMA) approach, where the segment is divided into a number of *mini-slots* of equal length δ. A *dynamic slot* is a logical entity, which can consist of one or more mini-slots, depending on whether data is sent on the slot and how much data is sent. Once a dynamic slot is assigned, if no data is sent in a communication cycle, only one mini-slot is consumed. If data is to be sent, a number of mini-slots are occupied to accommodate the data. The dynamic slots can be represented as $\mathscr{S}_{DYN} = \{N_s + 1, \ldots, N_s + N_{ms}\}$, where N_{ms} is the number of mini-slots.

The communication cycles are organized as sequences of 64 cycles. In a sequence, each communication cycle is indexed by a *cycle counter* which counts from 0 to 63 and is then set to 0. A FlexRay schedule corresponding to the message $f_{x,i}$ can be defined as $\Theta_{x,i} = (s_{x,i}, q_{x,i}, r_{x,i})$, where $s_{x,i}$ represents the slot number, $q_{x,i}$ represents the *base cycle*, and $r_{x,i}$ represents the *repetition rate*. Here, the subscript $x \in \{s, c\}$ where s or c respectively identifies sensor or control message. The subscript i identifies the control application C_i it constitutes. Here, the repetition rate $r_{x,i}$ is the number of communication cycles that elapse between two consecutive transmissions of the same frame and takes the value $r_{x,i} \in \{2^n | n \in \{0, \ldots, 6\}\}$. The base cycle $q_{x,i}$ is the offset of the cycle counter. The sequence of 64 communication cycles and some examples of FlexRay schedules are shown in Fig. 37.4. Here, the FlexRay Version 3.0.1 [3] is considered, where *slot multiplexing* among different ECUs is allowed. It means that a particular slot $s \in \mathscr{S}_{ST} \cup \mathscr{S}_{DYN}$ can be assigned to different ECUs in different communication cycles. Further consider all messages are sent over the static segment of the FlexRay bus, i.e., on the static slots. The starting and ending time of the k^{th} instance ($k \in \mathbb{Z}^*$) of the FlexRay schedule Θ_i, which are denoted respectively as $\bar{t}(\Theta_i, k)$ and $\tilde{t}(\Theta_i, k)$, can be defined as

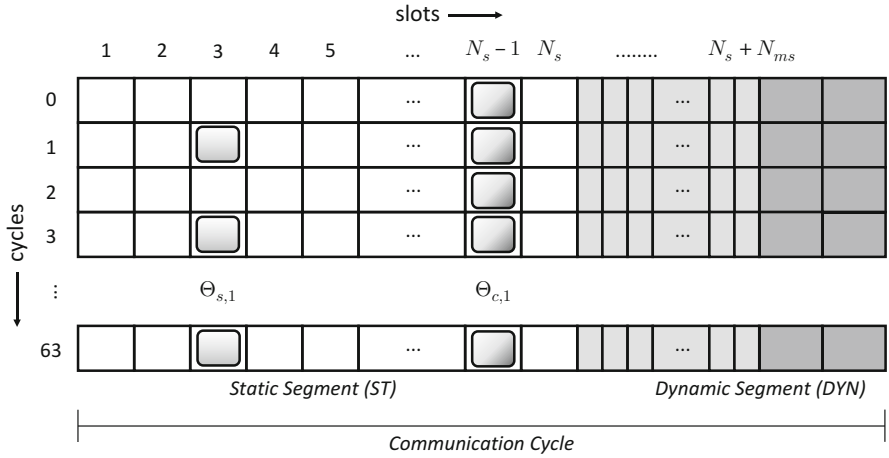


Fig. 37.4 An example of FlexRay schedules

$$\begin{aligned} \bar{t}(\Theta_{x,i}, k) &= q_{x,i} T_{bus} + k r_{x,i} T_{bus} + (s_{x,i} - 1)\Delta, \\ \tilde{t}(\Theta_{x,i}, k) &= q_{x,i} T_{bus} + k r_{x,i} T_{bus} + s_{x,i} \Delta. \end{aligned} \tag{37.22}$$

For FlexRay time-triggered communication, the bus utilization can be defined as the percentage of bandwidth of the static segment that is allocated to the control applications. This can be represented as the percentage of static slots allocated to the control applications in 64 consecutive communication cycles. In this case, the smaller the value of U, the better is the resource efficiency as more number of slots can be left vacant for use by other non-control applications. Now, let Γ denote the set of all FlexRay schedules allocated to the control applications on the static segment, where $\Theta_{x,i} \in \Gamma$; then the bus utilization U can be defined as

$$U = \frac{100}{64 N_s} \sum_{\Theta_{x,i} \in \Gamma} \frac{64}{r_{x,i}}, \tag{37.23}$$

where $64N_s$ is the total number of static slots in 64 consecutive communication cycles. Here, $r_{x,i}$ represents the repetition rate of the message $f_{x,i}$, and therefore, $\frac{64}{r_{x,i}}$ represents the number of static slot allocated to the message $f_{x,i}$ in 64 consecutive communication cycles.

Control performance: Depending on the specific requirements of the control application, one of the two performance metric discussed in Sect. 37.2.2.1 can be used. For a specific control application C_i, J_i depends both on the sampling period h_i and the control gains $K_{aug,i}$ and $F_{aug,i}$. In both the performance metrics, smaller value of J implies better control performance. In a system consisting of multiple control applications with different plant models and performance metrics,

it is required to normalize the control performance in order to compare and combine them. Each control system C_i with control performance J_i must satisfy some control performance requirement J_i^r defined by the user. Thus, the control performance can be normalized as follows:

$$J_i^n = \frac{100 \cdot J_i}{J_i^r} \tag{37.24}$$

and thus the overall control performance of a set of control applications $\mathscr{C}$ can be represented as a weighted sum

$$J_o = \sum_{C_i \in \mathscr{C}} w_i J_i^n, \tag{37.25}$$

where w_i stands for the weight and $\sum_i w_i = 1$.

Cooptimization problem: The cooptimization problem boils down to finding a set of parameters for each $C_i \in \mathscr{C}$, which can be denoted as $par_i = \{\tau_{s,i}, \tau_{c,i}, \tau_{a,i}, \Theta_{s,i}, \Theta_{c,i}, h_i, K_{aug,i}, F_{aug,i}\}$, while optimizing the total FlexRay bus utilization and the overall control performance given by Eqs. (37.23) and (37.25), respectively. Here, the control parameters of C_i can be further defined as $par_i^c = \{h_i, K_{aug,i}, F_{aug,i}\}$ and similarly the embedded platform parameters as $par_i^s = \{\tau_{s,i}, \tau_{c,i}, \tau_{a,i}, \Theta_{s,i}, \Theta_{c,i}\}$, where $par_i = par_i^s \cup par_i^c$. The parameter set of the whole system is represented as $\mathscr{P}$, where $par_i \in \mathscr{P}$.

37.3.2 The Codesign Approach

37.3.2.1 Design Flow

Figure 37.5 shows the design flow of the codesign approach. The whole design process is divided into two stages. In the first stage, for each control application, possible controllers that optimize the control performance at different sampling periods are synthesized and the results are recorded in a look-up table. In the second stage, the cooptimization stage, both the control and the platform parameters are synthesized based on the constraints, objectives, and the look-up tables obtained in the first stage. Here, a bi-objective optimization problem is formulated, and a customized approach is used to generate a Pareto front of the two objectives considered. In this stage, the fact that the bus utilization objective U can only take selected discrete values is exploited, and therefore, for each of those values, a nested two-layered optimization technique is employed to find a feasible set of parameters that represents a Pareto point and optimizes the control performance or to prove that a corresponding Pareto point is not possible. Here, Layer 1 tries to find a set of values of sampling periods corresponding to the set of control applications such that it can represent a Pareto point and it optimizes the overall system control performance for a given value of bus utilization. Then, Layer 2 tries to find a

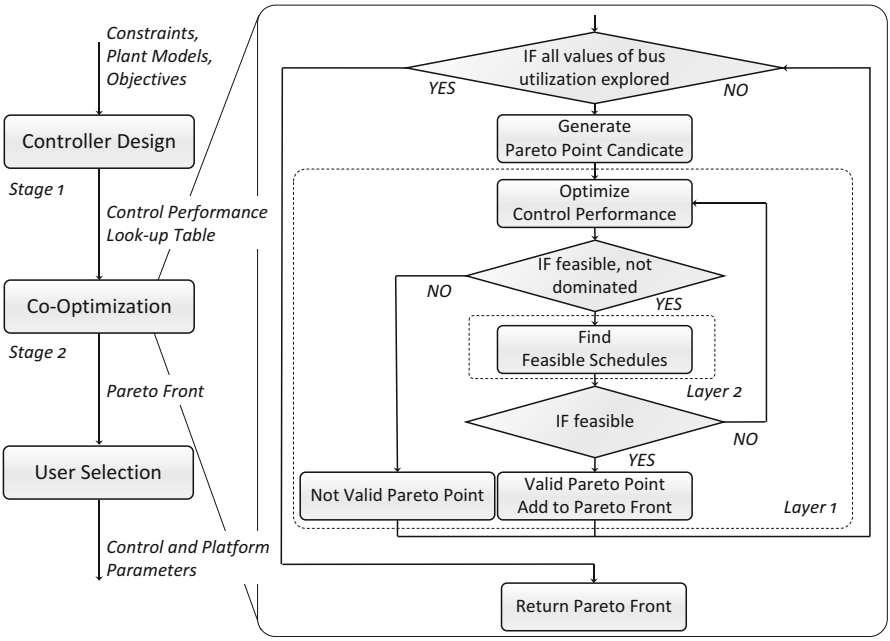


Fig. 37.5 Design flow of the cooptimization approach

feasible schedule set (by solving a constraint programming problem) and control gains (from the look-up tables) corresponding to the sampling period values of the control applications determined in Layer 1. The nested two-layered optimization technique is discussed in further detail in Sect. 37.3.2.4. Based on the Pareto front thus obtained, the designer can select one set of parameters that is the most suitable for the overall design requirements. The control design stage and the cooptimization stage of this approach will be explained in detail in the following sections.

37.3.2.2 Controller Design

Besides the control plant model, the performance J_i of the control application C_i depends mainly on three factors: (i) the sampling period h_i, (ii) the sensor-to-actuator delay d_i, and (iii) the control gains $K_{aug,i}$ and $F_{aug,i}$. Depending on each combination of the sampling period and delay, a set of optimal control gains needs to be designed. Here, consider schedules for the control tasks and the messages leading to the case where the delay equals to the sampling period, i.e., $d_i = h_i$. This would reduce the dimensions of the design space from all three factors (i)–(iii) to only (i) and (iii), thus reducing the complexity and enhancing the scalability. It should be noted that this approach can be easily adapted to other cases with a fixed delay value (e.g., $d_i = D_i$, where D_i is a constant and $D_i \leq h_i$) or a delay value proportional to the sampling period (e.g., $d_i = \psi h_i$, where $\psi \leq 1$). With $d_i = h_i$, the closed-loop system experiences one sampling period delay, and the pole-placement method

reported in Sect. 37.2 can be used for such delayed system. To the best of our knowledge, there is no standard closed-form optimal control design framework that can be directly applied in such a delayed system. Therefore, the PSO-based optimal pole-placement technique described in Sect. 37.2.2.2 is employed, which can be quite computationally costly for higher-order control plants. However, making use of the fact that the sampling period can only take discrete values, the design space can be pruned. Since each control application C_i is implemented by the tasks $\tau_{s,i}$, $\tau_{c,i}$, and $\tau_{a,i}$ and messages $f_{s,i}$, $f_{c,i}$, there is a dependency between the sampling period h_i and the repetition rate of the messages $r_{s,i}$, $r_{c,i}$, which can be represented as

$$h_i = r_{s,i} T_{bus} = r_{c,i} T_{bus}. \tag{37.26}$$

Due to the fact that $r_{s,i}$, $r_{c,i}$ can only take discrete values in $\{2^k | k \in \{0, \ldots, 6\}\}$, the choice of h_i is also constrained to the corresponding discrete values.

Denote the control performance as $J_i = f(h_i, K_{aug,i}, F_{aug,i})$. Then the control performance at each discrete value $h_i^k = 2^k T_{bus}$ of the sampling period can be represented as $J_i(h_i^k) = g(K_{aug,i}^k, F_{aug,i}^k)$. The purpose of the controller design step is to determine the control gains for each possible value of the sampling period that optimizes the control performance. Employing the optimal pole-placement technique, determine the set of control gains $K_{aug,i}^{k*}$, $F_{aug,i}^{k*}$ that optimizes the control performance to G_i^{k*} at sampling period h_i^k, then represent the optimal control performance at h_i^k as $J_i^*(h_i^k) = G_i^{k*}$. The control design problem can be translated into the problem of finding for each discrete value h_i^k, a set of gains $K_{aug,i}^{k*}$, $F_{aug,i}^{k*}$ that optimizes the control performance $J_i(h_i^k)$ to the value of G_i^{k*}.

After this stage, a look-up table for each control application C_i can be formulated where for each of the possible sampling period h_i^k an optimal control performance G_i^{k*} corresponding to the control gains $K_{aug,i}^{k*}$, $F_{aug,i}^{k*}$ can be assigned. In the cooptimization stage, this set of tables will be used to formulate the objective of overall control performance.

37.3.2.3 Optimization Problem Formulation

The system constraints for the FlexRay-based ECU system are well studied and discussed in [23, 29, 35]. Here, we will state the majority of the constraints formulated there.

(C1) Sampling period constraint: The tasks and messages of a control application must have the same period of repetition which is also the sampling period of the system. This constraint can be formulated as

$$\forall C_i \in \mathscr{C}, x \in \{s, c, a\}, y \in \{s, c\}, \qquad P_{x,i} = r_{y,i} T_{bus} = h_i. \tag{37.27}$$

(C2) Data-flow constraint: In a control application, all task executions and message transmissions must be in correct temporal order, as illustrated in Fig. 37.3. This can be formulated as set of constraints as

$$\begin{aligned}
&\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\tau_{s,i}, k) + \epsilon < \bar{t}(\Theta_{s,i}, k),\\
&\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\theta_{s,i}, k) < \bar{t}(\tau_{c,i}, k) - \epsilon,\\
&\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\tau_{c,i}, k) + \epsilon < \bar{t}(\Theta_{c,i}, k),\\
&\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\theta_{c,i}, k) < \bar{t}(\tau_{a,i}, k) - \epsilon.
\end{aligned} \tag{37.28}$$

(C3) Sensor-to-actuator delay constraint: The constraint stating that the sensor-to-actuator delay for the control applications is equal to exactly one sampling period can be formulated as

$$\forall k \in \mathbb{Z}^*, C_i \in \mathscr{C}, \quad \tilde{t}(\tau_{a,i}, k+1) - \bar{t}(\tau_{s,i}, k) = h_i. \tag{37.29}$$

(C4) Non-overlapping task constraint: In a time-triggered non-preemptive scheduling scheme as considered in this paper, when more than one task is mapped on an ECU, they must be scheduled in such a way that they do not overlap. This can be formulated as a constraint as

$$\begin{aligned}
&\forall \quad C_i, C_j \in \mathscr{C}, \quad x, y \in \{s, c, a\}, \quad p_k \in \mathbb{P}\\
&\forall \quad \{m \in \mathbb{Z}^* | 0 \le m < lcm(P_{x,i}, P_{y,j})/P_{x,i}\},\\
&\quad \{n \in \mathbb{Z}^* | 0 \le n < lcm(P_{x,i}, P_{y,j})/P_{y,j}\}\\
&\textbf{if} \quad \tau_{x,i}, \tau_{y,j} \in \mathscr{T}_{p_k} \quad \textbf{then} \quad \tilde{t}(\tau_{x,i}, m) + \epsilon \cdot \mathbb{1}(x \in \{s, c\}) < \bar{t}(\tau_{y,j}, n)\\
&\quad - \epsilon \cdot \mathbb{1}(y \in \{c, a\})\\
&\textbf{or} \quad \tilde{t}(\tau_{y,j}, n) + \epsilon \cdot \mathbb{1}(y \in \{s, c\}) < \bar{t}(\tau_{x,i}, m) - \epsilon \cdot \mathbb{1}(x \in \{c, a\}),
\end{aligned} \tag{37.30}$$

where $\mathscr{T}_{p_k}$ denotes the set of all tasks mapped on ECU E_k. $\mathbb{1}(.)$ is the indicator function and takes the value of 1 if the input is true and 0 if otherwise.

(C5) Nonoverlapping message constraint: FlexRay messages must be scheduled in such a way that no two messages share the same slot in the same cycle. This constraint can be established as

$$\begin{aligned}
&\forall \quad C_i, C_j \in \mathscr{C}, \quad x, y \in \{s, c\}\\
&\forall \{n \in \mathbb{Z}^* | 0 \le n < max(r_{x,i}, r_{y,j})/r_{x,i}\},\\
&\quad \{m \in \mathbb{Z}^* | 0 \le m < max(r_{x,i}, r_{y,j})/r_{y,j}\},\\
&\textbf{if} \quad s_{x,i} == s_{y,j} \quad \textbf{then} \quad q_{x,i} + n r_{x,i} \neq q_{y,j} + m r_{y,j}.
\end{aligned} \tag{37.31}$$

(C6) FlexRay scheduling constraint: Taking into consideration the scheduling constraints imposed by the FlexRay protocol, it is required to constrain $s_{x,i}$ and $q_{x,i}$ as

$$\begin{aligned} &\forall \;\; C_i \in \mathscr{C}, \quad x \in \{s, c\}, \quad 1 \leq s_{x,i} \leq N_s \\ &\forall \;\; C_i \in \mathscr{C}, \quad x \in \{s, c\}, \quad 0 \leq q_{x,i} < r_{x,i}. \end{aligned} \tag{37.32}$$

In addition, the bus utilization U is constrained by the total number of static slots available in 64 communication cycles.

$$U \leq 100. \tag{37.33}$$

(C7) ECU scheduling constraint: On ECUs, for task schedules, consider

$$\forall \;\; C_i \in \mathscr{C}, \quad x \in \{s, c, a\}, \quad 0 \leq O_{x,i} + E_{x,i} < P_{x,i}. \tag{37.34}$$

Moreover, the ECU load cannot be more than 100%.

$$\forall p_k \in \mathbb{P}, x \in \{s, c, a\}, \sum_{\tau_{x,i} \in \mathscr{T}_{p_k}} \frac{E_{x,i} + \epsilon + \epsilon \cdot \mathbb{1}(x \in \{c\})}{P_{x,i}} \leq 1., \tag{37.35}$$

(C8) Performance constraint: For each control system C_i with sampling period h_i, user specifies a control performance requirement J_i^r. As mentioned in Sect. 37.3.2.2, a look-up table for each control system is developed which contains the performance of seven possible controllers corresponding to seven possible sampling periods. Therefore, the domain of h_i, denoted as $dom[h_i]$ is constrained according to control performance requirement as

$$\forall k \in \{0, 1, \ldots 6\}, \quad J_i^*(h_i^k) \leq J_i^r \iff h_i^k \in dom[h_i]. \tag{37.36}$$

Now, let J_i represent the control performance of C_i. Therefore,

$$h_i == 2^k T_{bus} \iff J_i == J_i^*(h_i^k). \tag{37.37}$$

As the objectives for the optimization problem, the overall system control performance and the bus utilization are considered.

(O1) Overall system control performance:

$$J_o = \sum_{C_i \in \mathscr{C}} w_i J_i^n = \sum_{C_i \in \mathscr{C}} w_i \sum_k \mu_{i,k} J_i^{n*}(h_i^k), \tag{37.38}$$

where $\mu_{i,k}$ are binary variables satisfying $\sum_k \mu_{i,k} = 1$ and $J_i^{n*}(h_i^k)$ represents the normalized optimal control performance of C_i at h_i^k, which can be formulated as

$$J_i^{n*}(h_i^k) = \frac{100 J_i^*(h_i^k)}{J_i^r}. \tag{37.39}$$

(O2) Bus utilization: The bus utilization in this case can be defined as

$$U = \frac{100}{64 N_s} \sum_{C_i \in \mathscr{C}} \left(\frac{64}{r_{s,i}} + \frac{64}{r_{c,i}} \right) = \frac{100}{64 N_s} \sum_{C_i \in \mathscr{C}} \frac{128 T_{bus}}{h_i}. \tag{37.40}$$

The value of the bus utilization can only take certain discrete values and is bounded by the upper and lower limit U^+ and U^-, which can be expressed as

$$U^+ = \frac{100}{64 N_s} \sum_{C_i \in \mathscr{C}} \frac{128 T_{bus}}{\max\limits_{h_i \in dom[h_i]} (h_i)}, \qquad U^- = \frac{100}{64 N_s} \sum_{C_i \in \mathscr{C}} \frac{128 T_{bus}}{\min\limits_{h_i \in dom[h_i]} (h_i)}. \tag{37.41}$$

37.3.2.4 Multi-objective Optimization

As discussed above, the control and system codesign of the setting considered can be formulated as a constrained optimization problem with two objectives, namely, the bus utilization and overall control performance. In this case, the two design objectives are noticed to be often conflicting, and therefore, as discussed in ▸ Chap. 6, "Optimization Strategies in Design Space Exploration", a much more informative and designer-friendly cooptimization approach is to first generate a Pareto front, and let the designer explore the trade-off between the two objectives according to his customized preference.

▸ Chapters 6, "Optimization Strategies in Design Space Exploration", ▸ 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration", and ▸ 9, "Scenario-Based Design Space Exploration" have emphasized on hybrid optimization techniques to solve such a Design Space Exploration (DSE) problem. Such techniques depend heavily on problem characteristics, desired accuracy and scalability, etc. Consequently, for this problem, a customized hybrid optimization approach as shown in Fig. 37.5 is employed to obtain the desired Pareto front. Since the objective on bus utilization U is discrete and only takes a limited number of integers, first compute the maximum and minimum bus utilization U^+ and U^-, which bound the set of U. For each possible value of U from U^- to U^+, i.e., given the equality constraint on U, solve the optimization problem with J_o as the single objective and obtain a solution. The additional constraint is that J_o of this solution has to be better than J_o of the last solution (Pareto criterion), in order to ensure that all solutions are non-dominated. Therefore, the cooptimization problem with two objectives is turned into a series of single-objective optimization problems, where each may generate a Pareto point on the Pareto front.

Popular approaches like Mixed Integer Linear Programming (MILP) or meta-heuristic methods cannot be applied directly to solve each of the single-objective optimization problems. However, considering that some decision variables only appear in constraints, but are not related to the objective, a nested two-layered

technique is employed to solve each of the problems. On Layer 1, the outer layer, consider only constraint (C8) and an equality constraint on bus utilization U translated from (O2), and optimize the (O1). Decision variables related to the objectives, i.e., the sampling periods, are determined. On Layer 2, the inner layer, the remaining decision variables are synthesized satisfying the constraints (C1)–(C7) while substituting the values of sampling periods based on the results of Layer 1. This process is iterative in the way that if the synthesis fails in Layer 2, the algorithm goes back to Layer 1 for the next best solution until Pareto criterion is satisfied. This optimization technique ensures optimality and also efficiency.

37.3.3 Case Study

In the case study, five control applications denoted as $\mathscr{C} = \{C_1, C_2, C_3, C_4, C_5\}$ are considered. For each of the control applications, a plant model derived from the automotive domain is used. C_1 to C_5 represent respectively the DC motor speed control (DCM), servo motor position control (DCP), the electronic braking control (EBC), the car suspension (CSS), and the adaptive cruise control (ACC). The hardware platform consists of three ECUs $\{E_1, E_2, E_3\}$ connected by FlexRay bus. Tables 37.1 and 37.2 show the task mappings and FlexRay bus configuration, respectively.

Figure 37.6 shows the results of the normalized optimal control performance for each control application as the sampling period increases. The thick red dashed line in the plot shows the normalized required performance for all the control applications (i.e., 100%). Only the points below the red line meet the design requirement for performance, and only these points will be considered in the following cooptimization stage. The Pareto front of the whole system in the case

Table 37.1 Task mapping

ECUs	Tasks
E_1	$\tau_{s,1}$, $\tau_{c,2}$, $\tau_{a,3}$,
	$\tau_{a,4}$, $\tau_{c,5}$
E_2	$\tau_{a,1}$, $\tau_{s,2}$, $\tau_{c,3}$,
	$\tau_{s,4}$, $\tau_{s,5}$
E_3	$\tau_{c,1}$, $\tau_{a,2}$, $\tau_{s,3}$,
	$\tau_{c,4}$, $\tau_{a,5}$

Table 37.2 FlexRay bus configuration

Bus parameters	Values
Bus speed	10 Mbps
T_{bus}	5 ms
N	25
M	237
Δ	100 ms
δ	10 ms

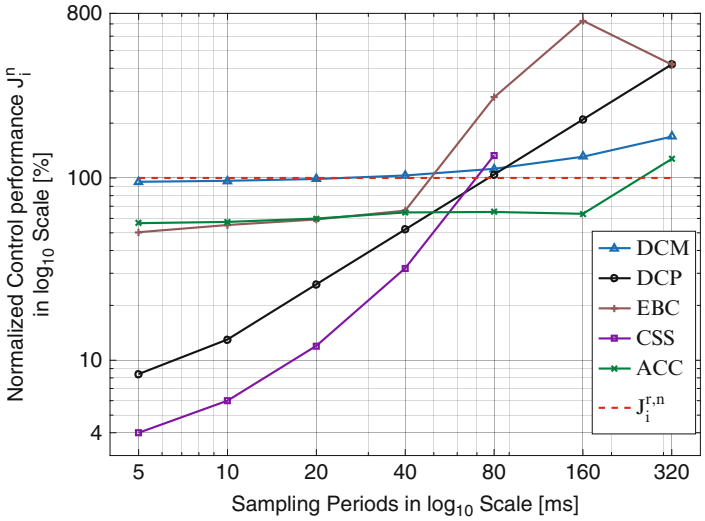


Fig. 37.6 Control performance

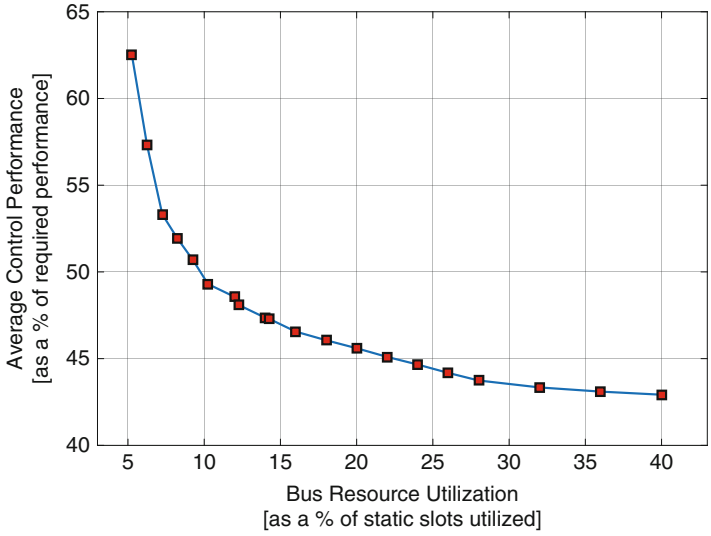


Fig. 37.7 Pareto front

study obtained in the cooptimization stage is shown in Fig. 37.7. The value of the bus utilization ranges from 5.25% to 40% of the bus bandwidth in the static segment. The value of the control performance varies on an average from 42.92% to 62.54% of the required value for each control application. It should be noted that for the control performance defined here, the smaller the value, the better the performance. It is obvious that there is a large freedom among these viable design points.

37.4 Memory-Aware Control/Architecture Codesign

While the memory-aware optimization of embedded software has been discussed in ▸ Chap. 26, "Memory-Aware Optimization of Embedded Software for Multiple Objectives", in this section, how to exploit the instruction cache reuse to improve the control performance is shown. Given a collection of control applications (e.g., C_1, C_2, C_3) on one processing unit, it is conventional to run the control loops of them in a round-robin fashion ($C_1, C_2, C_3, C_1, C_2, C_3, \cdots$). Since the programs for different control applications are different, the cache in this process is frequently refreshed. This results in poor cache reuse and long WCET. In order to address this issue, a memory-aware sampling order for the control applications can be applied, using which cache reuse is improved and the WCET of each application is reduced. In particular, we study a nonuniform sampling scheme, where the control loop of each application is consecutively run multiple times – in order to increase the cache reuse – before moving on to the next application (e.g., $C_1, C_1, C_1, C_2, C_2, C_2, C_3, C_3, C_3, \cdots$). As illustrated in Fig. 37.8, where $C_i(j)$ denotes the jth execution of the control application C_i, before the first execution $C_i(1)$, the cache is either empty (i.e., cold cache) or filled with instructions from other applications that are not used by C_i (equivalent to cold cache). The WCET of $C_i(1)$ can be computed by a number of existing standard techniques [9,37,38]. Before the second execution $C_i(2)$, the instructions in the cache are from the same application C_i and thus can be reused. This results in more cache hits and hence shorter WCET. Depending on which path the program takes, the amount of WCET reduction varies. Therefore, a technique is required to compute the guaranteed WCET reduction of $C_i(2)$ and $C_i(3)$, independent of the path taken, which will be presented later in this section. Control parameters of the systems, such as sampling periods and sensor-to-actuator delays, are derived from the WCET results. A controller must be tailored for the memory-aware nonuniform sampling orders, in order to improve the control performance. In summary, two main techniques are required and explained as

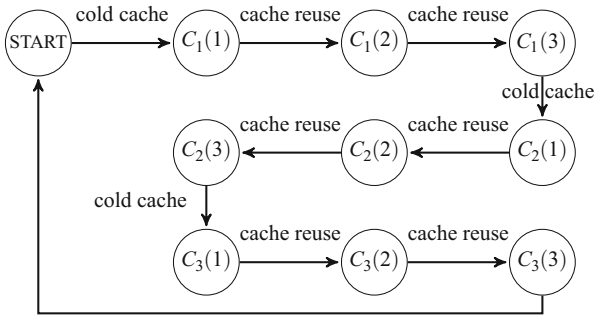


Fig. 37.8 An example memory-aware sampling order with three applications. Each application is consecutively executed three times. After the first execution $C_i(1)$, some instructions in the cache can be reused, and thus the WCETs of the following two executions are shortened

follows: (i) cache analysis to compute the guaranteed WCET reduction between two consecutive executions of one program and (ii) controller design for the nonuniform sampling.

37.4.1 Cache Analysis for Consecutive Executions of a Control Application

As discussed in Sect. 37.1, a two-level memory hierarchy – cache and main memory – is considered. More information about the memory architecture can be found in ▸ Chap. 13, "Memory Architectures". There are N_c cache lines, denoted as $CL = \{c_0, c_1, \ldots, c_{N_c-1}\}$, and the main memory has N_m blocks, denoted as $M = \{m_0, m_1, \ldots, m_{N_m-1}\}$. Each memory block is mapped to a fixed cache line. An example is shown in Fig. 37.9 for the illustration purpose, where there are four cache lines and five memory blocks. A basic block is a straight-line sequence of code with only one entry point and one exit point. This restriction makes a basic block

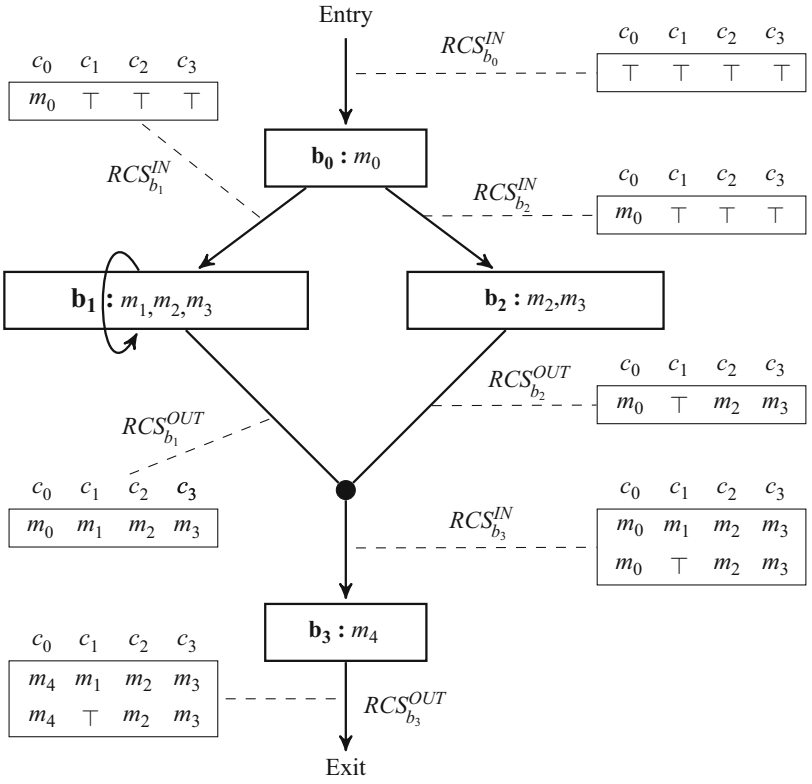


Fig. 37.9 A motivational example for cache analysis. Five memory blocks are mapped to four cache lines. Memory blocks executed by each basic block are shown. RCS^{IN} and RCS^{OUT} in the initialization phase are illustrated

highly amenable for program analysis. The presented Control-Flow Graph (CFG) in Fig. 37.9, consisting of four basic blocks $B = \{b_0, b_1, b_2, b_3\}$, has all the three key elements of a control program, i.e., sequential basic blocks, branches, and a loop. Therefore, it is suitable for illustrating our cache analysis technique.

There are three key terms in cache analysis that are described as follows:

- *Cache States*: A cache state cs is described as a vector of N_c elements. Each element $cs[i]$, where $i \in \{0, 1, \ldots, N_c - 1\}$, represents the memory block in the cache line c_i. When the cache line c_i holds the memory block m_j, where $j \in \{0, 1, \ldots, N_m - 1\}$, $cs[i] = m_j$. If c_i is empty, it is denoted as $cs[i] = \perp$. If the memory block in c_i is unknown, it is denoted as $cs[i] = \top$. CS is the set of all possible cache states.
- *Reaching Cache States (RCS)*: RCS of a basic block b_k, denoted as RCS_{b_k}, is the set of all possible cache states when b_k is reached via any incoming path.
- *Live Cache States (LCS)*: LCS of a basic block b_k, denoted as LCS_{b_k}, is the set of all possible first memory references to cache lines at b_k via any outgoing path.

Since our focus is on WCET reduction between two consecutive executions of C_i, it is necessary to compute RCS of the exit point in the first execution of C_i and LCS of the entry point in the second execution of C_i. By comparing all possible pairs of cache states, the guaranteed number of cache hits, and thus WCET reduction can be calculated. In the following, computation of RCS and LCS is firstly discussed.

In RCS computation, gen_{b_k} is firstly defined as the cache state describing the last executed memory block in every cache line for the basic block b_k. Assuming that b_0 in Fig. 37.9 executes m_0 and then m_4, instead of only m_0, the last executed memory block in c_0 is m_4. Therefore, gen_{b_0} is $[m_4, \perp, \perp, \perp]$. For the example in Fig. 37.9,

$$\begin{aligned} gen_{b_0} &= [m_0, \perp, \perp, \perp], \quad & gen_{b_1} &= [\perp, m_1, m_2, m_3], \\ gen_{b_2} &= [\perp, \perp, m_2, m_3], \quad & gen_{b_3} &= [m_4, \perp, \perp, \perp]. \end{aligned} \tag{37.42}$$

There are two equations involved in the RCS computation that calculate RCS^{IN} and RCS^{OUT}, where RCS^{IN} of a basic block b_k is the RCS before b_k is executed and RCS^{OUT} is the set of all possible cache states after b_k is executed. First, $RCS^{OUT}_{b_k}$ can be calculated from $RCS^{IN}_{b_k}$ as

$$RCS^{OUT}_{b_k} = \{\mathscr{T}(b_k, cs) | cs \in RCS^{IN}_{b_k}\}, \tag{37.43}$$

where $\mathscr{T}$ is a transfer function defined as follows: For any cache state $cs \in CS$ and basic block $b_k \in B$, there is a cache state $cs' = \mathscr{T}(b_k, cs)$, where for any cache line $c_i \in CL$ and $i \in \{0, 1, \ldots, N_c - 1\}$,

Table 37.3 RCS computation for the motivational example

	Basic block	RCS^{IN}	RCS^{OUT}
Initialization	b_0	$\{[\top, \top, \top, \top]\}$	$\{[m_0, \top, \top, \top]\}$
	b_1	$\{[m_0, \top, \top, \top]\}$	$\{[m_0, m_1, m_2, m_3]\}$
	b_2	$\{[m_0, \top, \top, \top]\}$	$\{[m_0, \top, m_2, m_3]\}$
	b_3	$\{[m_0, m_1, m_2, m_3], [m_0, \top, m_2, m_3]\}$	$\{[m_4, m_1, m_2, m_3], [m_4, \top, m_2, m_3]\}$
Fixed-point	b_0	$\{[\top, \top, \top, \top]\}$	$\{[m_0, \top, \top, \top]\}$
	b_1	$\{[m_0, \top, \top, \top], [m_0, m_1, m_2, m_3]\}$	$\{[m_0, m_1, m_2, m_3]\}$
	b_2	$\{[m_0, \top, \top, \top]\}$	$\{[m_0, \top, m_2, m_3]\}$
	b_3	$\{[m_0, m_1, m_2, m_3], [m_0, \top, m_2, m_3]\}$	$\{[m_4, m_1, m_2, m_3], [m_4, \top, m_2, m_3]\}$

$$cs'[i] = \begin{cases} cs[i] & : if\ gen_{b_k}[i] = \perp; \\ gen_{b_k}[i] & : otherwise. \end{cases} \tag{37.44}$$

$RCS^{IN}_{b_k}$ can be calculated as

$$RCS^{IN}_{b_k} = \bigcup_{p \in predecessor(b_k)} RCS^{OUT}_{p}, \tag{37.45}$$

where $predecessor(b_k)$ is the set of all immediate predecessors of b_k.

The RCS computation is composed of two phases: initialization and fixed-point computation. As illustrated with the example in Fig. 37.9, the initialization phase starts from the entry basic block b_0 with $RCS^{IN}_{b_0} = \{[\top, \top, \top, \top]\}$. The element is $\top$ since our analysis is independent of the program executed before b_0. According to (37.43), $RCS^{OUT}_{b_0}$ is calculated to be $\{[m_0, \top, \top, \top]\}$. Since b_0 is the only immediate predecessor of b_2, $RCS^{IN}_{b_2}$ is equal to $RCS^{OUT}_{b_0}$ based on (37.45). Due to the self-loop, b_1 has both itself and b_0 as immediate predecessors. However, since $RCS^{OUT}_{b_1}$ has not been initialized yet, $RCS^{IN}_{b_1}$ is equal to $RCS^{OUT}_{b_0}$. In the same manner, $RCS^{OUT}_{b_1}$, $RCS^{OUT}_{b_2}$, $RCS^{IN}_{b_3}$, and $RCS^{OUT}_{b_3}$ can be computed, following the program flow as shown both in Fig. 37.9 and Table 37.3. The initialization phase is completed once all basic blocks have been visited. The next phase is fixed-point computation. RCS^{IN} and RCS^{OUT} of all basic blocks are computed iteratively with (37.45) and (37.43). This phase is terminated once the fixed point is reached, i.e., RCS^{IN} and RCS^{OUT} of all basic blocks remain unchanged. Let the program RCS be the RCS^{OUT} of the exit basic block, i.e., $RCS = RCS^{OUT}_{b_3}$. Results are reported in Table 37.3.

The LCS computation can be done in a similar fashion. gen_{b_k} is defined as the cache state describing the first executed memory block in every cache line for the basic block b_k. Taking the same assumption when defining gen_{b_k} for RCS computation that b_0 in Fig. 37.9 executes m_0 and then m_4, instead of only m_0, the

Table 37.4 LCS computation for the motivational example

	Basic block	LCS^{IN}	LCS^{OUT}
Initialization	b_3	$\{[\top, \top, \top, \top]\}$	$\{[m_4, \top, \top, \top]\}$
	b_2	$\{[m_4, \top, \top, \top]\}$	$\{[m_4, \top, m_2, m_3]\}$
	b_1	$\{[m_4, \top, \top, \top]\}$	$\{[m_4, m_1, m_2, m_3]\}$
	b_0	$\{[m_4, m_1, m_2, m_3], [m_4, \top, m_2, m_3]\}$	$\{[m_0, m_1, m_2, m_3], [m_0, \top, m_2, m_3]\}$
Fixed-point	b_3	$\{[\top, \top, \top, \top]\}$	$\{[m_4, \top, \top, \top]\}$
	b_2	$\{[m_4, \top, \top, \top]\}$	$\{[m_4, \top, m_2, m_3]\}$
	b_1	$\{[m_4, \top, \top, \top], [m_4, m_1, m_2, m_3]\}$	$\{[m_4, m_1, m_2, m_3]\}$
	b_0	$\{[m_4, m_1, m_2, m_3], [m_4, \top, m_2, m_3]\}$	$\{[m_0, m_1, m_2, m_3], [m_0, \top, m_2, m_3]\}$

first executed memory block in c_0 is m_0. Therefore, gen_{b_0} is $[m_0, \perp, \perp, \perp]$. LCS^{IN} of a basic block b_k is the LCS after b_k is executed and can be derived from

$$LCS_{b_k}^{IN} = \bigcup_{s \in successor(b_k)} LCS_s^{OUT}, \tag{37.46}$$

where $successor(b_k)$ is the set of all immediate successors of b_k. LCS^{OUT} of b_k is the LCS before b_k is executed with

$$LCS_{b_k}^{OUT} = \{\mathscr{T}(b_k, cs) | cs \in LCS_{b_k}^{IN}\}. \tag{37.47}$$

LCS computation also comprises two phases of initialization and fixed-point computation. The only difference is that the initialization phase starts from the exit basic block and ends in the entry basic block. Detailed results for the motivational example are reported in Table 37.4. Let the program LCS be the LCS^{OUT} of the entry basic block, i.e., $LCS = LCS_{b_0}^{OUT}$. It is noted that since the presented cache analysis technique is based on the fixed-point computation over the program CFG, it inherently handles loop structures in the CFG.

Conceptually, the program RCS is the set of all possible cache states after the program finishes execution by any execution path, and the program LCS is the set of all cache states, where each cache state contains memory blocks that may be firstly referenced after the program starts execution, for any execution path to follow. Both RCS and LCS could contain multiple cache states. Each pair with one cache state cs from the program RCS and one cache state cs' from the program LCS represents one possible execution path between the two consecutive executions. For any cache line c_i in a pair, if $cs[i]$ is equal to $cs'[i]$ and they are not equal to $\top$, then there is certainly a hit and thus WCET reduction. Whether there is a hit for a particular cache line can be determined by the function $\mathscr{H}$ defined as follows:
$\forall cs \in CS$, $cs' \in CS$ and $c_i \in CL$, where $i \in \{0, 1, \ldots, N_c - 1\}$,

$$\mathscr{H}(cs, cs', c_i) = \begin{cases} 1 : if\ cs[i] = cs'[i] \wedge cs[i] \neq \perp; \\ 0 : otherwise. \end{cases} \tag{37.48}$$

The number of hits can be counted with the function $\mathscr{HT}$ defined as $\forall cs \in CS$ and $cs' \in CS$,

$$\mathscr{HT}(cs, cs') = \sum_{i=0}^{N_c-1} \mathscr{H}(cs, cs', c_i). \tag{37.49}$$

The guaranteed number of hits among all possibilities is calculated as

$$\mathscr{G}(RCS, LCS) = \min_{cs \in RCS, cs' \in LCS} (\mathscr{HT}(cs, cs')). \tag{37.50}$$

Given that the main memory access time and the cache access time are respectively t_m and t_c, the guaranteed WCET reduction is computed as

$$\begin{aligned} \bar{E}^g &= \mathscr{G}(RCS, LCS) \times (t_m - t_c) \\ &\approx \mathscr{G}(RCS, LCS) \times t_m, \end{aligned} \tag{37.51}$$

where the approximation can be taken if $t_c \ll t_m$.

For the motivational example, there are two cache states in RCS ($RCS_{b_3}^{OUT}$) and two cache states in LCS ($LCS_{b_0}^{OUT}$). In total, there are four pairs, and the number of hits is calculated to be 3, 2, 2, and 2 with (37.49). Taking one of them as an example, $\mathscr{HT}([m_4, m_1, m_2, m_3], [m_0, m_1, m_2, m_3]) = 3$. Therefore, the guaranteed number of hits is 2 according to (37.50), no matter which path the program takes. From (37.51), the guaranteed WCET reduction is $2 \times (t_m - t_c)$, or approximately $2 \times t_m$, when $t_c \ll t_m$. It is noted that this result is obtained from the small example used for illustration. More WCET reduction for larger realistic programs can be expected.

Note that the direct-mapped cache (i.e., one-way set-associative cache) is assumed in Fig. 37.9. The presented technique can be adapted to handle set-associative cache. For example, considering fully associative cache, when computing $RCS_{b_3}^{OUT}$ from $RCS_{b_3}^{IN}$, the memory block m_4 can be loaded to any cache line, which gives $RCS_{b_3}^{OUT}$ five more cache states, i.e., $[m_0, m_4, m_2, m_3]$, $[m_0, m_1, m_4, m_3]$, $[m_0, \top, m_4, m_3]$, $[m_0, m_1, m_2, m_4]$, and $[m_0, \top, m_2, m_4]$. From this, it can be observed that the number of cache states in RCS and LCS is larger for set-associative cache, which means that the guaranteed WCET reduction could be smaller. Details can be found in [27]. Using the cache analysis technique presented in this section, together with standard WCET analysis approaches, the effective WCET of $C_i(2)$ and subsequent executions of C_i can be derived. Shorter WCET leads to smaller sampling period of the control system, which will be shown next.

37.4.2 Control Parameter Derivation

We explore the relationship between WCET results and control parameters of two example sampling schemes. S1 is the conventional memory-oblivious scheme and summarized as follows:

$$\begin{aligned} &C_1(1) \to C_2(1) \to C_3(1) \to C_1(2) \to C_2(2) \to \\ &C_3(2) \to C_1(3) \to C_2(3) \to C_3(3) \to \cdots . \end{aligned} \tag{37.52}$$

There is no cache reuse in S1 in the worst case, considering that different control applications typically have different instructions to execute. In other words, when $C_i(j)$ starts execution, all instructions of C_i need to be brought into the cache from the main memory. Therefore,

$$E_i^{wc}(1) = E_i^{wc}(2) = \cdots = E_i^{wc}, \tag{37.53}$$

where $E_i^{wc}(j)$ is the WCET of the jth execution for C_i. The WCET of the application C_i is denoted by E_i^{wc}, since all executions of the same application have equal WCET. Clearly, all control applications run with a uniform sampling period of

$$h = \sum_{i=1,2,3} E_i^{wc}. \tag{37.54}$$

Moreover, the sensor-to-actuator delay, which is defined to be the duration between measuring the system state $x(t)$ and applying the control input $u(t)$, is given by

$$d_i = E_i^{wc}. \tag{37.55}$$

It can be seen that a safe estimation of WCET, which can be done with standard static analysis techniques [37], is very important. If the actual execution time is longer than the computed WCET, the correct control input will not be ready when the actuation is supposed to occur. The consequence could be severe degradation of control performance. This is not acceptable especially for safety-critical control applications.

S2 is an example of memory-aware sampling order as shown in Fig. 37.8:

$$\begin{aligned} &C_1(1) \to C_1(2) \to C_1(3) \to C_2(1) \to C_2(2) \to \\ &C_2(3) \to C_3(1) \to C_3(2) \to C_3(3) \to \cdots . \end{aligned} \tag{37.56}$$

The effective WCET taking into account the cache reuse is denoted with $\bar{E}_i^{wc}(j)$. From the above discussion,

$\forall i \in \{1, 2, 3\}$,

$$\bar{E}_i^{wc}(1) = E_i^{wc}, \tag{37.57}$$

since there is no cache reuse for the first execution of every application $C_i(1)$. $\bar{E}_i^{wc}(2)$ and $\bar{E}_i^{wc}(3)$ are shorter than $\bar{E}_i^{wc}(1)$ due to cache reuse. The amounts of cache reuse are the same for $C_i(2)$ and $C_i(3)$ in the worst case. Denoting the guaranteed WCET reduction as $\bar{E}_i^g$,

$\forall i \in \{1, 2, 3\}$,

$$\bar{E}_i^{wc}(2) = \bar{E}_i^{wc}(3) = \bar{E}_i^{wc}(1) - \bar{E}_i^g. \tag{37.58}$$

From these varying WCETs, the sampling periods of all three applications can be calculated. Taking C_1 as an example, there are three sampling periods $h_1(1)$, $h_1(2)$, and $h_1(3)$, which repeat themselves periodically:

$$h_1(1) = \bar{E}_1^{wc}(1), \quad h_1(2) = \bar{E}_1^{wc}(2), \quad h_1(3) = \bar{E}_1^{wc}(3) + \Delta, \tag{37.59}$$

where Δ is computed as

$$\Delta = \sum_{i=2,3} \sum_{j=1,2,3} \bar{E}_i^{wc}(j). \tag{37.60}$$

Similar derivation can be done for C_2 and C_3. The average sampling period of an application h_{avg} is

$$h_{\text{avg}} = \frac{\sum_{i=1,2,3} \sum_{j=1,2,3} \bar{E}_i^{wc}(j)}{3} < h. \tag{37.61}$$

According to (37.57) and (37.58),

$$h_{\text{avg}} < \frac{\sum_{i=1,2,3} 3 \times E_i^{wc}}{3}. \tag{37.62}$$

From (37.54),

$$h_{\text{avg}} < h. \tag{37.63}$$

Moreover, the corresponding sensor-to-actuator delay $d_i(j)$ also varies with cache reuse as

$\forall i \in \{1, 2, 3\}$,

$$d_i(1) = h_i(1) = \bar{E}_i^{wc}(1), \quad d_i(2) = h_i(2) = \bar{E}_i^{wc}(2), \quad d_i(3) = \bar{E}_i^{wc}(3). \tag{37.64}$$

As all control parameters have been derived, it can be observed that the sampling period $h_i(j)$ of a control application is nonuniform for the memory-aware scheme. The average sampling period of S2 is shorter than the uniform sampling period of S1 as shown in (37.61), due to WCET reduction resulting from cache reuse. The sensor-to-actuator delay $d_i(j)$ varies as shown in (37.64). The next task is to develop a controller design method to exploit shortened nonuniform sampling periods and achieve better control performance. For the uniform sampling scheme, the sensor-to-actuator delay d_i is shorter than the sampling period h. Therefore, the technique reported in Sect. 37.2 is used. Details of the controller design technique considering the nonuniform sampling are reported in the next section.

37.4.3 Case Study

Here a commonly used processing unit, equipped with a processor, on-chip memory as cache and flash as main memory is considered, shown in Fig. 37.1 More about the flash memory has been discussed in ▶ Chap. 14, "Emerging and Nonvolatile Memory". As a case study, three control applications are considered. C_1 is position control of a servo motor. C_2 is speed control of a DC motor. C_3 is control of an electronic wedge brake system. All three control applications run on the same processor. The processor clock frequency is 20 MHz. The cache is set to have 128 cache lines and each cache line is 16 bytes. When there is a cache hit, it takes 1 clock cycle to fetch the instruction, and when there is a cache miss, it takes 100 clock cycles. WCET results are reported in Table 37.5. Sampling periods of the two sampling orders S1 and S2 are shown in Table 37.6. Control performances of three applications under S1 and S2 are presented in Table 37.7, where the settling time is taken as the performance metric. As an example, the system output responses of C_1 under both S1 and S2 are presented in Fig. 37.10. The control task considered is to change the system output (i.e., the angular position of the servo motor) from 0 to 0.3 rad. From the above experimental results, it can be clearly seen that the

Table 37.5 WCET results with and without cache reuse for all three control applications

Application	WCET without cache reuse	WCET with cache reuse	Reduction percentage
C_1	907.55 μs	452.15 μs	50.18%
C_2	645.25 μs	175.00 μs	72.88%
C_3	749.15 μs	234.35 μs	68.72%

Table 37.6 Comparison of sampling periods between S1 and S2 for all three control applications. The reduction percentage is computed according to the average sampling period

Application	Sampling periods in S1	Sampling periods in S2	Reduction percentage
C_1	2302 μs	452 μs – 452 μs – 3121 μs	42%
C_2	2302 μs	175 μs – 175 μs – 3675 μs	42%
C_3	2302 μs	234 μs – 234 μs – 3557 μs	42%

Table 37.7 Control performances for all three applications under S1 and S2

Application	C_1	C_2	C_3
Settling time for S1	31.2 ms	26.8 ms	25.2 ms
Settling time for S2	21.5 ms	21.1 ms	20.4 ms
Control performance improvement of S2 compared to S1	31.1%	21.3%	19.0%

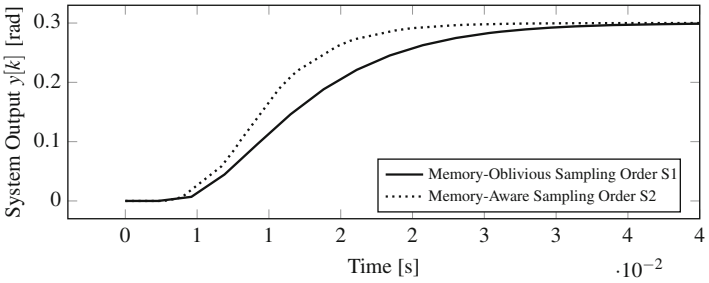


Fig. 37.10 Control system output of $\mathscr{C}_1$ under S1 and S2

memory-aware sampling order reduces the WCETs and sampling periods. With the controller design method tailored for nonuniform sampling, control performances are significantly improved.

37.5 Computation-Aware Control/Architecture Codesign

In this section, we show how to use a multirate controller to reduce the processor utilization of a control application, while still fulfilling the control performance requirement and system constraints. More information about the application-specific processors can be found in ▸ Chap. 12, "Application-Specific Processors".

37.5.1 Time-Triggered Operating System

As an example, ERCOSek with the OSEK/VDX standard [1] is considered, which specifies the basic properties of an OS to be used in the automotive domain. In general, as an OSEK/VDX OS, ERCOSek supports preemptive fixed-priority scheduling. That is, priorities are assigned to applications, and at any point in time, the task with the highest priority among all active ones is executed. On ERCOSek, tasks can be triggered by events (e.g., interrupts, alarms, etc.) or by time. In the time-triggered scheme, each application gets released and is allowed to access the processor periodically. There are various periods of release times and each application is assigned one. Every time an application is released, its task gets the chance to be executed. A time table containing all the periodic release times within

Table 37.8 Example of an ERCOSek time table

Time	Release
0 ms	Applications with periods of 2 ms/5 ms/10 ms
2 ms	Applications with the period of 2 ms
4 ms	Applications with the period of 2 ms
5 ms	Applications with the period of 5 ms
6 ms	Applications with the period of 2 ms
8 ms	Applications with the period of 2 ms
10 ms	**Repeat actions at 0 ms**

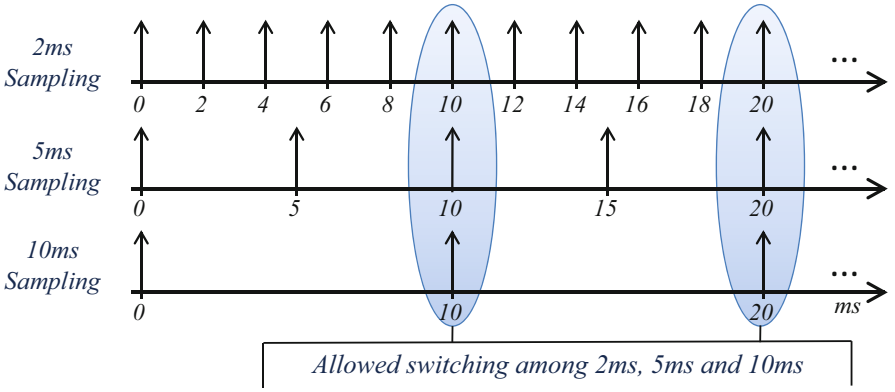


Fig. 37.11 Allowed switching instants among multiple sampling periods

the alleged hyperperiod (i.e., the minimum common multiple of all periods) needs to be configured. An example with a set of three periods 2, 5, and 10 ms is illustrated in Table 37.8. The hyperperiod is equal to 10 ms and the time table repeats itself every 10 ms by resetting the timer. Independent of the triggering mode (i.e., be it event or time triggered), the assigned priority will still determine the execution order of tasks. In the time-triggered scheme, a higher priority is typically assigned to the application released with a shorter period, since this generally results in a more efficient use of the processor.

Assuming the set of available periods restricted by ERCOSek to be ϕ, control applications have to be sampled with one period or a combination of multiple periods from ϕ. In the latter case, switching between two sampling periods can only occur at the common multiplier of them, as illustrated in Fig. 37.11, considering three sampling periods 2, 5, and 10 ms. Often, the optimal sampling period for a control application does not belong to the set ϕ. The simple and straightforward method used in practice is to select the largest sampling period in ϕ that is smaller than the optimal one. This results in a higher processor load, which is another important design aspect. Denoting E_i^{wc} to be the WCET of a control application C_i, if the uniform sampling period is T, the processor load for C_i is

$$L_i = \frac{E_i^{wc}}{T}. \tag{37.65}$$

The upper bound on the load of any processor is 1. Considering a single processor p,

$$\sum_{\{i|C_i \text{ runs on } p\}} L_i \leq 1. \tag{37.66}$$

Clearly, increasing the sampling period of a control application decreases its processor load and thus potentially enables more applications to be integrated on the processor.

37.5.2 Multirate Closed-Loop Dynamics

We consider a multirate controller switching between multiple sampling periods in ϕ, toward achieving an average sampling period close to the optimal one. The cyclic sequence of sampling periods for a control application defines a schedule S:

$$S = \{T_1, T_2, T_3, \ldots, T_N\}, \tag{37.67}$$

where $\forall j \in \{1, 2, \ldots, N\},\ T_j \in \phi$. It implies the sequence of sampling periods as

$$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_N \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_N \rightarrow repeat$$

Following the assumption in (37.65) that the WCET of C_i is E_i^{wc}, the processor load for C_i over S is

$$L_i = \frac{NE_i^{wc}}{\sum_{j=1}^{N} T_j}. \tag{37.68}$$

Dictated by the schedule S, N systems switch cyclically in a deterministic fashion. When the sampling period $t_{k+1} - t_k = T_{k,j}$, the dynamics is

$$x[k+1] = A_d(T_{k,j})x[k] + B_d(T_{k,j})K_j x[k] + B_d(T_{k,j})F_j r. \tag{37.69}$$

The controller design needs to be performance oriented, and the key is to compute the feedback gain K_j for each system with pole placement, based on which the static feedforward gain F_j can be derived with (37.10).

Referring to Fig. 37.12, after the first sampling interval of a switching cycle,

$$x[k+1] = A_d(T_{k,1})x[k] + B_d(T_{k,1})\hat{K}_1 x[k] + B_d(T_{k,1})F_1 r. \tag{37.70}$$

It is noted that K_1 is the feedback gain based on the most recent system state $x[k]$ and used to compute the control input. $\hat{K}_1$ is the equivalent feedback gain based on the starting system state $x[k]$ of a switching cycle. In this case that only one

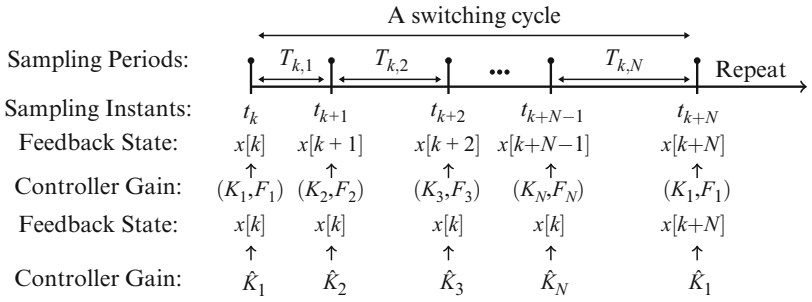


Fig. 37.12 Cyclically switched linear systems

sampling period is considered, $\hat{K}_1 = K_1$. The feedforward gain F_1, which is related to K_1, is also based on the most recent system state and used to compute the control input. The closed-loop system matrix is denoted as $A_{cl,1}$ and

$$A_{cl,1} = A_d(T_{k,1}) + B_d(T_{k,1})\hat{K}_1. \tag{37.71}$$

$\hat{K}_1$ can be designed by pole placement. Poles to place are eigenvalues of $A_{cl,1}$. F_1 is computed as per (37.10).

After the second sampling interval,

$$x[k+2] = A_d(T_{k,2})x[k+1] + B_d(T_{k,2})K_2x[k+1] + B_d(T_{k,2})F_2r. \tag{37.72}$$

To consider the overall dynamics of the first two sampling periods, the relation between $x[k+2]$ and $x[k]$ can be derived as

$$\begin{aligned} x[k+2] = {} & A_d(T_{k,2})A_{cl,1}x[k] + B_d(T_{k,2})K_2A_{cl,1}x[k] \\ & + (A_d(T_{k,2}) + B_d(T_{k,2})K_2)B_d(T_{k,1})F_1r + B_d(T_{k,2})F_2r. \end{aligned} \tag{37.73}$$

Let

$$\hat{K}_2 = K_2A_{cl,1}, \tag{37.74}$$

and (37.73) becomes

$$\begin{aligned} x[k+2] = {} & A_d(T_{k,2})A_{cl,1}x[k] + B_d(T_{k,2})\hat{K}_2x[k] \\ & + (A_d(T_{k,2}) + B_d(T_{k,2})K_2)B_d(T_{k,1})F_1r + B_d(T_{k,2})F_2r. \end{aligned} \tag{37.75}$$

Similar to (37.71),

$$A_{cl,2} = A_d(T_{k,2})A_{cl,1} + B_d(T_{k,2})\hat{K}_2. \tag{37.76}$$

It is noted that (37.75) has the same form as (37.70). $\hat{K}_2$ can be designed by pole placement and K_2 is derived with (37.74), as long as $A_{cl,1}$ is non-singular. Poles to place are eigenvalues of $A_{cl,2}$. F_2 is computed as per (37.10). Continuing the above analysis,

$\forall j \in \{1, 2, \ldots, N\}$

$$A_{cl,j} = A_d(T_{k,j})A_{cl,j-1} + B_d(T_{k,j})\hat{K}_j, \tag{37.77}$$

and $A_{cl,0} = \mathbf{I}$ can be defined. $\hat{K}_j$ can be designed by pole placement. Poles to place are eigenvalues of $A_{cl,j}$. As long as $A_{cl,j-1}$ is non-singular, K_j is derived by

$$K_j = \hat{K}_j A_{cl,j-1}^{-1}. \tag{37.78}$$

F_j is computed as per (37.10).

Here the sensor-to-actuator delay is approximately equal to the WCET of the executed control program. Since the control law is computed during the design phase, such a control program generally has a short WCET. The sensor-to-actuator delay is often negligible compared to the sampling periods given by the OS. In general, when the sensor-to-actuator delay of a control task is large compared to the sampling periods (e.g., in the memory-aware controller design of Sect. 37.4, where the sampling periods are directly constrained by WCETs), our proposed controller design technique can be extended to consider the delayed control input with a number of methods reported in the literature [24].

An optimization problem for the pole placement can be formulated as presented in Sect. 37.2.2.2. The number of dimensions in the decision space is nN – the number of states of the application multiplied by the number of sampling periods in the schedule. The optimization objective is the settling time. Absolute values of all poles have to be less than unity to ensure system stability and larger than 0 to make all $A_{cl,j}$ non-singular.

Optimization strategies for design space exploration have been discussed in ▸ Chap. 6, "Optimization Strategies in Design Space Exploration". In this section, to solve one optimization problem, the PSO algorithm is run multiple times with the same number of particles, and we do not set the limit on the number of iterations. If the objective value variation of the solution points from these runs exceeds a certain threshold (e.g., 1%), the number of particles is increased. Considering the stochastic nature of PSO, it is very likely that the optimal point has been found when multiple runs generate similar objective values. It is noted that if the number of sampling periods in the schedule is very large, which makes the number of dimensions in the decision space very large, this method aiming to ensure optimality can be computationally expensive. In this case, the number of particles and iterations has to be limited, resulting in a compromise in optimality.

Table 37.9 Settling times of three schedules

Schedule	Settling time [ms]	Requirement
$S1 = \{5\,\text{ms}\}$	253.69	Violated
$S2 = \{2\,\text{ms}\}$	110.44	Satisfied
$S3 = \{2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 5\,\text{ms}, 5\,\text{ms}\}$	128.6 ms	Satisfied

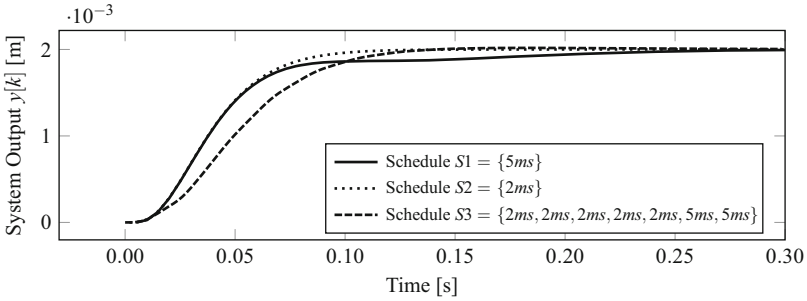


Fig. 37.13 System outputs of different schedules

37.5.3 Case Study

The presented multirate controller design technique is evaluated with an Electro-Mechanical Brake (EMB) system used in automobiles. It can be modeled as (37.1) with five system states. The control input is the voltage output of the onboard battery and thus cannot exceed 12 V. Different controllers require different battery voltage output profiles, and only those that respect the input constraint are possible to implement. The constraint on the settling time is 150 ms. In the optimization, the settling time is still treated as the objective to minimize and check the optimal solution against this requirement. The WCET of the control program is 0.2 ms. The control task is to change the system output (i.e., the position of the lever) from 0 to 2 mm. The set of available sampling periods offered by ERCOSek is

$$\phi = \{1\,\text{ms}, 2\,\text{ms}, 5\,\text{ms}, 10\,\text{ms}, 20\,\text{ms}, 50\,\text{ms}, 100\,\text{ms}, 200\,\text{ms}, 500\,\text{ms}, 1\,\text{s}\}. \tag{37.79}$$

As shown in Table 37.9 and Fig. 37.13, the schedule $S1 = \{5ms\}$ cannot meet the settling time requirement. The largest sampling period smaller than 5 ms in ϕ is 2 ms. The schedule $S2 = \{2\,\text{ms}\}$ is able to fulfill all the requirements. According to (37.65), the processor load of $S2$ is 0.1. Then a schedule switching between 2 ms and 5 ms is considered, $S3 = \{2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 2\,\text{ms}, 5\,\text{ms}, 5\,\text{ms}\}$. This sequence of sampling periods satisfies the OS requirement. The multirate controller is designed as discussed earlier in this section. The WCET (0.2 ms) is much shorter than the sampling periods (2 ms, 5 ms), and thus we neglect the sensor-to-actuator

delay. $S3$ has a slightly longer settling time than $S2$, but still fulfills the requirement. According to (37.68), the processor load is 0.07, achieving a 30% reduction compared to $S2$.

37.6 Conclusion

In this chapter, a basic introduction into the subject of control/architecture codesign in the context of cyber-physical systems is provided. The control/architecture codesign is an emerging field of research, where the design of control parameters and embedded platform parameters are integrated in a holistic approach to reduce conservativeness and achieve more efficient design of embedded control systems. As the size and the complexity of the cyber-physical systems increase, resource-efficient design has become one of the most important aspects in this context. In this chapter, the motivation is firstly explained, and some basic concepts of the control/architecture codesign are introduced. In addition, a brief summary on the type of resources that can be considered in the codesign approaches is provided. Then three examples of state-of-art codesign approaches, targeting respectively at communication-aware design, memory-aware design, and computation-aware design, are used to illustrate the basic thinking behind the control/architecture codesign. In Sect. 37.3, a cooptimization framework is explained to codesign control and platform parameters by solving a constraint-based multi-objective optimization problem. This framework considers two objectives, namely, the resource utilization and the overall control performance, and generates a Pareto front depicting the trade-off options between the two objectives. In Sect. 37.4, how to exploit the instruction cache reuse in a memory-aware sampling order to improve the control performance is shown. Cache analysis is used to compute the guaranteed WCET reduction between two consecutive executions of one control program. Control parameters are derived based on the WCET results. The controller design is tailored for the nonuniform sampling scheme. In Sect. 37.5, the OS constraint that only a limited set of sampling periods are provided is considered. It is shown how a multirate controller is used to reduce the processor utilization of a control application, while still fulfilling the control performance requirement and system constraints. The control/architecture codesign is, of course, a relatively new and open research field, and thus the state-of-art approaches are certainly not limited to the ones shown in this chapter. There are also some other research directions in this context that can be explored. For example, power consumption is quite an important design factor, and thus power-aware codesign methods could potentially lead to more power-efficient designs. Furthermore, safety and fault tolerance are also important factors in cyber-physical systems which can also be considered in codesign methods. In addition, the three approaches shown in this chapter address individually a single resource. If the complexity of the problem due to many design dimensions can be tackled, it would be interesting to try to address simultaneous two or more resources in the codesign and thus offer an even greater freedom for design trade-offs.

References

1. OSEK/VDX operating system specification 2.2.3 (2005)
2. 664P7-1 aircraft data network, part 7, avionics full-duplex switched Ethernet network (2009)
3. The FlexRay communications system protocol specification, Version 3.0.1 (2010)
4. LIN specification package revision 2.2A (2010)
5. MOST specification rev. 3.0 E2 (2010)
6. AS6802 (2011) Time-triggered Ethernet
7. Infineon Product Brief XC2300B – Series (Accessed 12 May 2016). http://www.infineon.com/dgdl/Pb_XC2300B.pdf?fileId=db3a30432a7fedfc012ab3c3d7863706
8. Ackermann J, Utkin VI (1994) Sliding mode control design based on Ackermann's formula. In: Proceedings of the 33rd IEEE conference on decision and control, vol 4, Lake Buena Vista, pp 3622–3627. doi:10.1109/CDC.1994.411715
9. Andalam S, Sinha R, Roop P, Girault A, Reineke J (2013) Precise timing analysis for direct-mapped caches. In: 2013 50th ACM/EDAC/IEEE design automation conference (DAC), Austin, pp 1–10. doi:10.1145/2463209.2488917
10. Astrom KJ, Murray RM (2008) Feedback systems: an introduction for scientists and engineers. Princeton University Press, Princeton
11. Batcher KW, Walker RA (2008) Dynamic round-robin task scheduling to reduce cache misses for embedded systems. In: 2008 Design, automation and test in Europe, Munich, pp 260–263. doi:10.1109/DATE.2008.4484893
12. Bhave AY, Krogh BH (2008) Performance bounds on state-feedback controllers with network delay. In: 47th IEEE conference on decision and control, CDC 2008, Cancun, pp 4608–4613. doi:10.1109/CDC.2008.4739330
13. Bosch (1991) CAN Specification version 2.0. Stuttgart, Bosch
14. Castane R, Marti P, Velasco M, Cervin A, Henriksson D (2006) Resource management for control tasks based on the transient dynamics of closed-loop systems. In: 18th Euromicro conference on real-time systems (ECRTS'06), Dresden, pp 10, 182. doi:10.1109/ECRTS.2006.24
15. Cervin A, Velasco M, Marti P, Camacho A (2009) Optimal on-line sampling period assignment. Research report, Lund University and Technical University of Catalonia
16. Chang W, Chakraborty S (2016) Resource-aware automotive control systems design: a cyber-physical systems approach. Found Trends© Electron Design Autom 10(4):249–369. http://dx.doi.org/10.1561/1000000045
17. Charette RN (2009) This car runs on code. IEEE Spectrum. http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code
18. eCos. http://ecos.sourceware.org
19. Feiler PH (2003) Real-time application development with OSEK: a review of the OSEK standards. Technical report, Carnegie Mellon University
20. Gaid MEMB, Cela A, Hamam Y (2006) Optimal integrated control and scheduling of networked control systems with communication constraints: application to a car suspension system. IEEE Trans Control Syst Technol 14(4):776–787. doi:10.1109/TCST.2006.872504
21. Gaid MEMB, Cela A, Hamam Y, Ionete C (2006) Optimal scheduling of control tasks with state feedback resource allocation. In: 2006 American control conference, Minneapolis, pp 310–315. doi:10.1109/ACC.2006.1655373
22. Gloy N, Smith MD (1999) Procedure placement using temporal-ordering information. ACM Trans Program Lang Syst 21(5):977–1027. doi:10.1145/330249.330254
23. Goswami D, Lukasiewycz M, Schneider R, Chakraborty S (2012) Time-triggered implementations of mixed-criticality automotive software. In: 2012 Design, automation test in Europe conference exhibition (DATE), Dresden, pp 1227–1232. doi:10.1109/DATE.2012.6176680
24. Goswami D, Schneider R, Chakraborty S (2014) Relaxing signal delay constraints in distributed embedded controllers. IEEE Trans Control Syst Technol 22(6):2337–2345. doi:10.1109/TCST.2014.2301795

25. Henriksson D, Cervin A (2005) Optimal on-line sampling period assignment for real-time control tasks based on plant state information. In: Proceedings of the 44th IEEE conference on decision and control, Seville, pp 4469–4474. doi:10.1109/CDC.2005.1582866
26. Kalamationos J, Kaeli DR (1998) Temporal-based procedure reordering for improved instruction cache performance. In: Proceedings of fourth international symposium on high-performance computer architecture, Las Vegas, pp 244–253. doi:10.1109/HPCA.1998.650563
27. Kleinsorge JC, Falk H, Marwedel P (2011) A synergetic approach to accurate analysis of cache-related preemption delay. In: 2011 Proceedings of the international conference on embedded software (EMSOFT), Taipei, pp 329–338. doi:10.1145/2038642.2038693
28. Liu X, Chen X, Kong F (2015) Utilization control and optimization of real-time embedded systems. Found Trends© Electron Design Autom 9(3):211–307. http://dx.doi.org/10.1561/1000000042
29. Lukasiewycz M, GlaßM, Teich J, Milbredt P (2009) Flexray schedule optimization of the static segment. In: Proceedings of the 7th IEEE/ACM international conference on hardware/-software codesign and system synthesis, CODES+ISSS'09. ACM, New York, pp 363–372. doi:10.1145/1629435.1629485
30. Marti P, Lin C, Brandt SA, Velasco M, Fuertes JM (2004) Optimal state feedback based resource allocation for resource-constrained control tasks. In: Proceedings of 25th IEEE international on real-time systems symposium, Lisbon, pp 161–172. doi:10.1109/REAL.2004.39
31. Martí P, Lin C, Brandt SA, Velasco M, Fuertes JM (2009) Draco: efficient resource management for resource-constrained control tasks. IEEE Trans Comput 58(1):90–105. doi:10.1109/TC.2008.136
32. Pettis K, Hansen RC (1990) Profile guided code positioning. In: Proceedings of the ACM SIGPLAN 1990 conference on programming language design and implementation, PLDI'90. ACM, New York, pp 16–27. doi:10.1145/93542.93550
33. Pigan R, Metter M (2008) Automating with PROFINET, 2nd edn. Publicis Publishing, Erlangen
34. Samii S, Cervin A, Eles P, Peng Z (2009) Integrated scheduling and synthesis of control applications on distributed embedded systems. In: 2009 Design, automation test in Europe conference exhibition, Nice, pp 57–62. doi:10.1109/DATE.2009.5090633
35. Schneider R, Goswami D, Zafar S, Lukasiewycz M, Chakraborty S (2011) Constraint-driven synthesis and tool-support for flexray-based automotive control systems. In: Proceedings of the seventh IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES+ISSS'11. ACM, New York, pp 139–148. doi:10.1145/2039370.2039394
36. Sedighizadeh D, Masehian E (2009) Particle swarm optimization methods, taxonomy and applications. Int J Comput Theory Eng 1(4):486–502
37. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P (2008) The worst-case execution-time problem – overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3):36:1–36:53. doi:10.1145/1347375.1347389
38. Wilhelm R, Grund D, Reineke J, Schlickling M, Pister M, Ferdinand C (2009) Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans Comput Aided Des Integr Circuits Syst 28(7):966–978. doi:10.1109/T-CAD.2009.2013287
39. Zeng H, Natale MD, Ghosal A, Sangiovanni-Vincentelli A (2011) Schedule optimization of time-triggered systems communicating over the flexray static segment. IEEE Trans Ind Inf 7(1):1–17. doi:10.1109/TII.2010.2089465

Wireless Sensor Networks

38

Mihai Teodor Lazarescu and Luciano Lavagno

Abstract

Versatile and effective, Wireless Sensor Networks (WSNs) witness a continuous expansion of their application domains. Yet, their use is still hindered by issues such as reliability, lifetime, overall cost, design effort and multidisciplinary engineering knowledge, which often prove to be daunting for application domain experts. Several WSN design models, tools and techniques were proposed to solve these contrasting objectives, but no single comprehensive approach has emerged. With these criteria in mind we review several of the most representative ones, then we focus on two of the most effective hardware/software codesign flows. Both offer high-level design entry interfaces based on StateCharts. One allows manual module composition in a full application, and automates its mapping on a user-defined architecture for fast high-level design space exploration. The other flow automates module composition starting from the application specification and by reusing library modules. It can generate the hardware specification and the software to program and configure the WSN nodes. For these we show the typical use for the development of some representative applications, to evaluate their effectiveness.

Acronyms

6LoWPAN	IPv6 over Low Power Wireless Personal Area Network
ADC	Analog-to-Digital Converter
ADM	Abstract Design Module
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BOM	Bill of Materials
CAN	Controller Area Network

M.T. Lazarescu (✉) • L. Lavagno
Politecnico di Torino, Torino, Italy
e-mail: mihai.lazarescu@polito.it; luciano.lavagno@polito.it

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_38

CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DSML	Domain-Specific Modeling Language
EEPROM	Electrically Erasable Programmable Read-Only Memory
EMF	Eclipse Modeling-Framework
FSM	Finite-State Machine
GPIO	General-Purpose Input/Output-pin
GPRS	General Packet Radio Service
GPT	General-Purpose Timer
HAL	Hardware Abstraction Layer
I2C	Inter-Integrated Circuit
ICU	Input Capture Unit
ID	Identifier
I/O	Input/Output
IoT	Internet of Things
IP	Intellectual Property
ISR	Interrupt Service Routine
MAC	Media Access Control
MBD	Model-Based Design
MMC/SD	Multimedia/Secure Digital Card
NVIC	Nested Vectored Interrupt Controller
OS	Operating System
PWM	Pulse-Width Modulation
QoS	Quality of Service
RAM	Random-Access Memory
RC	Resistor-Capacitor
RFID	Radio-Frequency Identification
RF	Radio Frequency
RTC	Real-Time Clock
RTOS	Real-Time Operating System
SDC	Secure Digital Card
SPI	Serial Peripheral Interface
TCP/IP	Transmission Control Protocol/Internet Protocol
TWI	Two Wire Interface
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modeling Language
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
WSDL	Web Service Definition Language
WSN	Wireless Sensor Network
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Contents

38.1 Introduction

Wireless Sensor Networks (WSNs) already covers a broad range of applications in a variety of domains, which is continuously expanding, thanks to advances in research and technology. The range of requirements and problems that WSN designers must address are considerably more diversified today than when the Internet of Things (IoT) paradigm was coined by Kevin Ashton [3] more than 15 years ago. It is increasingly difficult to define "typical" application requirements for WSN hardware and software [31], since both must continuously adapt to very diverse WSN application requirements and operating conditions. Moreover, WSN platform reusability for a wide class of derived applications is becoming more important to lower development effort and time to delivery and to increase reliability.

Existing high-level WSN programming support of any kind is still seldom used for applications deployed in the real world [24]. System and application development and deployment using state-of-the art WSN technologies involve several different and complementary views, yet lacking mature separation of competencies between typical stakeholders and the various engineering disciplines that cover the WSN domain. For various practical reasons, WSN deployments are typically developed at a level very close to the embedded Operating System (OS), which often requires mastering a mix of low-level system and distributed protocol competencies that are seldom found among WSN application domain experts. Figure 38.1 shows how such development flows divert significant development efforts from the application logic, thus contributing to increase development time and cost and lowering overall reliability.

Another factor limiting WSN widespread use are the difficulties faced when porting an implementation to a different hardware platforms. They effectively

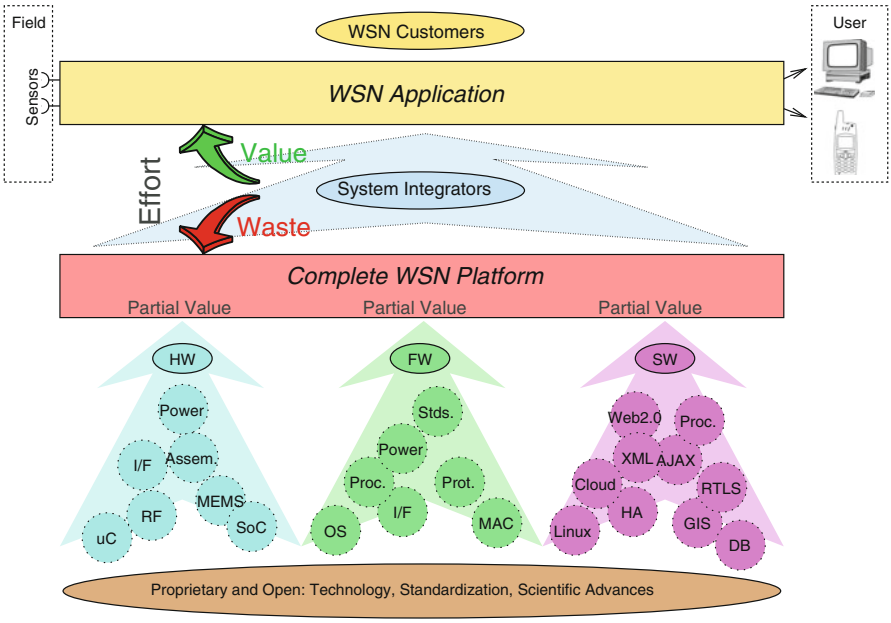


Fig. 38.1 Value flow for a WSN application and platform

reduce programmers' choices in terms of hardware platforms to those that are explicitly supported by each tool, which often is a narrow range.

High-level programming tools for WSN applications generally lack composability and the ability to be reused as building blocks. Code written according to a given programming abstraction can typically be used within a single framework because the collaboration between frameworks is still very limited. Although most frameworks are well suited for specific application domains, they can rarely be extended or composed with others. Consequently, their use is severely limited.

These are important aspects that limit the productivity of WSN application designers; lead to suboptimal or ineffective designs; increase the development time, cost, and risk; and reduce the WSN application reliability which, ultimately, increases maintenance cost.

In this context, we explore in detail two innovative tools for WSN application design. One is a model-based design framework for distributed WSN application development, design space exploration, network simulation (including hardware in the loop), and fast prototyping. It has a MATLAB Stateflow®-based [22] user-friendly interface for architecture-independent application entry, composition, and simulation and is based on an abstract service-based specification model for the application architecture. The framework can automatically map the application on several very different node platforms, and it can generate the simulation and implementation code for popular WSN simulators and operating systems. Hence, it hides most low-level implementation details from the developer in order to increase its usability by application domain experts and to allow fast high-level design space exploration.

The second toolset aims to further accelerate the development cycle and stimulate component reuse by automating the module selection and composition phases based on a high-level application specification. This flow does not make any assumption on the language or format of the behavior specification. Instead, it makes use of metadata that describe the behavior, interface and requirements of each library module, as well as the application specification. Besides simulation and implementation code, the system composition engine can generate nonfunctional requirements, such as a bill of materials and specifications for the target hardware or for the compilation toolchain.

Both frameworks are used to create several representative applications, in order to evaluate their effectiveness.

38.2 Past Work

As we discussed above, WSN design must satisfy two contrasting requirements:

1. Short design time with as little electronics and telecommunications expertise as possible. Many applications today are relatively small-scale and low-margin; hence, the nonrecurrent engineering costs should be kept as low as possible. Moreover, since application-specific aspects dominate both the difficulty and the gains, the most useful design resources are the domain experts.
2. Produce a very optimized implementation, especially for cost, power, and reliability. This is in order to minimize also the recurrent cost, both to procure the nodes, to deploy them, and to keep them running in the field.

The only way to satisfy both requirements is through design automation of some sort. As a result, the literature on WSN design is very rich of design aids, both in the form of design languages and their compilers, and in the form of support software, such as middleware, operating systems and so on. In this section, we review some of this work, in particular by referring to existing survey articles.

38.2.1 Programming Languages and Tools

A first excellent survey of programming languages and tools for WSNs, by Sugihara and Gupta, appeared in [34]. They classify devices and networks according to three orthogonal aspects:

1. Node power consumption, which is of course also related to the computational power, ranging from workstation-class computers, found typically at the "center" of the network, where the most complex elaboration takes place, to small battery-powered microcontrollers that have some ability to compute and route data, to tiny nodes (e.g., Radio-Frequency Identifications (RFIDs)) which scavenge energy from the environment. Most of the programming tools, including those

discussed in this chapter, apply to the middle group, where resources are scarce but computations are nontrivial.
2. Node observables, which range from just the data and an "address" (a node identifier), to the time and to the location of the observation. Satisfying real-time constraints and being able to reason about the location of the node add to the complexity of both the programming language and of the underlying middleware.
3. Size of the network, which ranges from tens to potentially millions of nodes, and which may require significant middleware support to ensure scalability.

At the lowest level of abstraction, namely, the individual node, the article lists several examples of operating systems and programming languages, classifying them according to the paradigm used for modularization: (1) message passing among statically "bound" software components, as in nesC [13] and TinyOS [17]; (2) dynamic association between messages and services, as in SNACK [14]; or (3) lightweight threads, as in Mantis OS [1]. Static binding requires the least resources but goes against flexibility and reconfigurability. Moreover, thread-based programming is more familiar to most developers but requires significantly larger memory resources. Virtual machines have also been considered in this context, although their portability advantages must be carefully weighed against their performance and energy cost.

The next layer of abstraction is group-level programming, in which application development, deployment, and maintenance are eased by the availability of programming constructs or APIs to reason about groups of nodes, based both on physical distance (neighborhood) and on logical properties. Making physical or logical location a first-class citizen of course enables much easier application development, since most WSN data processing must be aware of where the data itself originates.

Finally, the most abstract level considered in that article is the network level, where it describes:

1. database abstractions, where the network is viewed as a huge database, in which nodes and time instants play the role of rows, while kinds of sensed data play the role of columns, as in TinyDB [19]. User-level queries must be decomposed and distributed to physical nodes and radio channels so that Quality of Service (QoS) objectives, such as timeliness and energy consumption, are appropriately optimized.
2. macroprogramming approaches, which provide transparent mechanisms, e.g., to replicate and distribute data, so that the network is viewed as a single distributed computing platform, as in Kairos [15]. This in principle can provide the best optimization opportunities, allowing one to move both computation toward the data and vice versa, depending on the application constraints and requirements. But the huge optimization space makes low-level code generation an extremely challenging task.

Then the article classifies a large number of approaches based on the aspects listed above and evaluates them by their energy efficiency, scalability, failure resilience, and collaboration level (e.g., using centralized or distributed triangulation among nodes to establish node and data location).

Mottola and Picco provide another excellent survey of programming languages and tools for WSNs in [24]. They also include concrete examples to illustrate the key aspects of each listed approach. In addition to the space and time aspects that were discussed by [34], they also classify applications based on:

1. The goal, i.e., pure sensing or sensing and reacting (or actuation). The former leads naturally to a single or a few sinks, while the latter encourages distributed processing to improve resiliency and reduce communication costs.
2. The interaction pattern: many-to-one, often associated with sensing, one-to-many, often associated with reacting, or many-to-many.
3. The need to support mobility of at least some nodes (e.g., in cattle monitoring applications). This of course requires support for dynamic interaction and reconfiguration, at least at the network level but often also at the application level.

The article then proceeds to classify the *programming languages* based on aspects such as:

1. the scope of communication: (1) physical neighborhood, (2) multihop within a subset of nodes (again based on physical neighborhood or logical grouping), and (3) network-wide.
2. the addressing mechanism: (1) physical, or statically assigned, versus (2) logical, or dynamically assigned based on, e.g., current sensor readings or location.
3. whether communication is explicitly exposed, as in nesC, or implicit, as in TinyDB.
4. the scope of a computation: whether a single statement in the programming language can change the state of: (1) a single node, (2) a group of nodes, or (3) the entire network. Again, the latter offers more scope for optimization, and moves the burden from the programmer to the "compiler," but requires the development of more complex design tools.
5. the data access mode across nodes, via: (1) database abstractions, (2) shared remote variables, (3) code that migrates to find its data, or (4) explicit message passing. Database languages are easy to use, but lead to extremely complex low-level code generation issues when efficiency is important, as is often the case in energy-limited WSNs. Shared variables are also familiar to programmers but are very difficult to synchronize and use correctly. This is especially true in a setting where communication is slow and unreliable, and bandwidth is limited. Code migration can increase the longevity of networks, which can be often expensive to deploy. Code migration is best coupled with energy harvesting, because it can adversely impact battery life. Finally, explicit message passing gives most control to the programmer and is often the preferred choice for real-life deployments.

6. the programming paradigm: (1) imperative, (2) declarative (e.g., SQL-like or functional), or (3) hybrid, where a declarative language, which provides faster application development, can be extended with procedural mechanisms for the most performance- or energy- and power-critical aspects.

Then the article looks briefly at the architectural aspects, e.g.:

1. whether the approach supports only the application programmers (as is often the case with declarative languages and implicit communication) or can be used to build all layers of the software (which normally requires an imperative paradigm and explicit message passing).
2. the ability to access and tune lower layers (e.g., to allow cross-layer optimization of the protocol stack).

Finally, a very large number of different approaches are mapped and classified according to the criteria above.

The article concludes by outlining open areas for further research, such as:

1. tolerance to failures, which is essential for long-term real-life use in an often harsh environment.
2. ease of debugging, which is especially problematic when the nodes are already in the field.
3. real-world deployment, which is always needed to validate new ideas in realistic settings.
4. evaluation methodology, which suffers again from the lack of well-recognized benchmarks and of sensor data coming from real-world deployments.

38.2.2 Middleware and Operating System

Middleware and operating system are essential to support fast development and deployment of WSN software. Hence, they are the foundation (explicitly or implicitly) of all the approaches to WSN design that are outlined in the articles listed above. These two aspects are the direct focus of several survey articles. We will mention only a few of them.

First of all, Mohamed and Al-Jaroodi in [23] classify middleware types according to lines that are very similar to those mentioned above, namely, virtual machine, database, application-driven, and message-oriented. They list and discuss several aspects that still challenge effective deployment and use of middleware for WSNs (and of WSNs in general). These are, namely, (1) scarcity of hardware resources, (2) dynamic changes of network topology and size, (3) heterogeneity, (4) network lifetime, (5) application dependency, (6) security, (7) quality of service, and (8) integration with the broader Internet context.

From this list, they derive several key requirements that should be satisfied by the middleware, e.g.:

1. run-time support for service registration, discovery, and use. This enables dynamic adaptation to changes both of the network topology and of the applications themselves.
2. service transparency to client applications, in particular to hide the heterogeneity of the underlying network.
3. configurability to support a variety of QoS, security, and resource consumption requirements.
4. support for self-organization, in the presence of dynamic network changes due to mobility, addition and retirement of nodes, and so on.
5. interoperability with a variety of underlying devices and network protocols.
6. efficient handling of huge volumes of data.
7. support for security, QoS requirement management, and interoperability with other systems.

The article concludes by classifying and evaluating 15 middleware approaches according to these requirements and by discussing the opportunities for future work.

Along similar lines, Mottola and Picco in [25] provide an outlook into WSN middleware research. One very interesting comment in this article is that most WSN work, including almost all the approaches mentioned in the surveys above, conspicuously ignores the ZigBee industrial standard that specifies how applications can access the network stack and that is supported by several commercial node platforms. While this can be explained by the difficulty to tune and exploit a closed platform, at least the compatibility with its recommendations should be taken into account.

They also analyze the state of the art, including their own TeenyLIME environment [7], and outline open research challenges, such as:

1. supporting one-to-many and many-to-many abstractions, as well as mobility.
2. providing high-level abstractions for application developers, who are often domain experts, rather than electronics or telecommunications engineers, without forgetting network deployers and maintainers.
3. the need for flexibility and expressive power without losing efficiency.
4. support for cross-layer optimizations and interactions within the network stack, which is essential for simultaneous energy and performance optimization, and is seen as a key differentiator between WSNs and telecommunication networks.
5. the need to permit reliable and predictable implementations, since WSNs are embedded systems, which often implement safety-critical applications.
6. support for multiple, concurrent applications, sometimes with very different constraints. These may even have dynamic after-deployment installation and update requirements.
7. integration within broader systems, including of course the Internet, which would require a chapter in itself, especially for its industrial and transportation application areas.

Finally, they again stress the need for all research on WSNs (including the middleware) to be concretely demonstrated in real-world scenarios, not just with simulation results.

Dong et al. [9] provide a good summary of challenges for WSN OSs. The requirements that they pose are similar to those discussed for middleware but are at a lower level: small footprint, energy efficiency, reliability, real-time guarantees, reconfigurability, and programming convenience. First of all, they describe the main components of an OS for WSNs:

1. Task scheduling, which may be event-driven as in TinyOS or thread-based as in Mantis OS. As mentioned above, the former is more efficient, while the latter is more familiar to programmers.
2. Dynamic linking and loading, which adds a lot of flexibility to the network but has a cost in terms of complexity and overhead.
3. Memory management, in particular support for permanent storage, such as flash memory, and for dynamic memory allocation, which may be a problem in resource-constrained nodes.
4. Resource abstraction, to hide details of the underlying hardware and, in some cases, virtualize its access.
5. Sensor interfaces, which provide similar abstraction and virtualization capabilities for the more WSN-specific aspects of the hardware platform.
6. Networking stack, which is an essential part by definition of any WSN OS and may provide higher-level services that cross into the middleware domain.

Then the authors describe and compare several notable examples of WSN OSs, such as TinyOS [17], Contiki [10], SOS [16], Mantis OS [1], Nano-RK [11], RETOS [5], and LiteOS [4].

The classification is based on various aspects, such as (1) static or dynamic resource allocation, (2) event-driven versus multi-threaded scheduling, (3) monolithic or modular architecture, (4) networking support, (5) real-time support, (6), language support, (7) file system support, (8) reprogramming, and (9) remote debugging.

The last part of the article evaluates each approach with respect to the requirements that were defined at the beginning, and provides several recommendations to researchers interested in this domain, which range from keeping the design simple and flexible to considering hardware requirements, application needs, and development costs.

38.2.3 Model-Driven Design

Finally, we will mention three articles that are more specific to the topic of this chapter, namely, model-driven and component-based design of WSN applications.

Shimizu et al. [32] describe a model-driven methodology and tool to speed up design and optimization of WSN applications. Different from the approach

described later in this chapter, which focuses only on the Model-Based Design (MBD) of the node code, they define three different Domain-Specific Modeling Languages (DSMLs), respectively, for the network level, the group level, and the node level. Each DSML essentially offers a set of choices for key design parameters at the corresponding layer:

- The DSML for the network considers (1) data source nodes, (2) aggregation and fusion nodes, and (3) sink nodes. At this level, designers can choose how often sensors are sampled and how often data are transmitted toward the sink by each class of nodes.
- The DSML for the group (neighborhood) is similar, but at this level, designers can also choose (1) the network topology (e.g., tree or mesh), (2) the amount of in-network processing (aggregation and fusion), as well as (3) the geographical grouping.
- The DSML for the node considers (1) sampling tasks, (2) aggregation and fusion tasks, (3) sending and receiving tasks, and (4) sink tasks. Here, the designers can make choices on every aspect covered by the approach; thus, they have full customization capabilities.

The use of three DSMLs allows teams with different areas of expertise to hierarchically design and manage a large network, while retaining full control over the result. Automated code generation for simulation completes the flow.

While this approach exploits nicely the advantages of MBD, it is not clear how the user can define an application which cannot be generated simply by choosing appropriate values for the model parameters. In other words, it basically offers a single, albeit very parameterized, WSN "application," which can be customized to cover a broad range of requirements but is not (and most likely can never be) fully general.

In Sect. 38.3, we will present a design framework that is focused on modeling the code of the application tasks themselves and on smartly linking the tasks together at node level.

Taherkordi et al. [35] describe REMORA, a component-based model that is much more advanced than the basic static composition mechanism supported in TinyOS. For example, it includes the ability to dynamically deploy and connect components. Components and their interfaces are described in REMORA using an Extensible Markup Language (XML) format that covers (1) offered and required services that are activated through events, (2) the state that is retained by each component across invocations, and (3) the component (in a C-like language).

The event modeling mechanism in XML is more flexible than its TinyOS counterpart, allowing one to (1) specify event attributes, (2) distinguish between application events and OS events, (3) configure events, and (4) define if they are point-to-point or multicast.

The framework has a very low overhead with respect to Contiki, while providing significantly better encapsulation capabilities, and thus designer productivity, than bare bones multi-threading.

Finally, Compton et al. [6] survey semantics specifications for WSNs, i.e., the ontologies that can be used to describe the requirements of a network and allow a compositional design approach. This is very relevant for our methodology, which is based on an ontology to implement the component search, constrained composition, and parameter value selection capabilities.

The authors describe the capabilities of semantic sensor networks, including the ability to:

1. classify sensors according to functionality, type of output, or method of measurement.
2. find sensors than can perform some measurement.
3. collect data based on various criteria (spatial, temporal, . . .).
4. perform domain-specific inferences on low-level data.
5. react to specific inferred or measured events.

The article then lists 12 different ontologies, both general-purpose and application-specific (e.g., for marine sensors), and compares them in terms of the aspects of a WSN that each of them can describe. These aspects consider:

1. the logical aspects of each node and of the network as a whole, in terms of hierarchy, node identity, node software, deployment, configurations, history, and kind of processes it can support.
2. the physical aspects of each node, such as location, power supply, node platform, physical dimensions, and operating conditions.
3. the observations that each node can make, in terms of accuracy, frequency, response mechanism (periodic or event-triggered), and field of sensing.
4. the sensing domain, considering the measurement units, the features that are measured, and the time.

Finally, the authors summarize how ontologies are supported by various reasoning mechanisms. Later in this chapter, we will discuss an approach where ontology use is extended to describe both the functional and nonfunctional elements that compose WSN nodes in order to allow the automatic synthesis of both node hardware and software needed to support the application functions.

38.3 Model-Based WSN Application Design

The need to improve important metrics of WSN application development such as cost, time to market, lifetime and reliability, as well as its accessibility to domain application experts, can be satisfied using high-level design flows that support some degree of automation.

In Sect. 38.2, we reviewed several models, tools, and techniques that have been proposed in this regard. Although a significant variety of tools was proposed, no single comprehensive approach has emerged.

In this chapter, faithful to the principles of hardware/software codesign that are discussed in the entire book, we present an MBD framework that can speed up and facilitate application development, design space exploration, simulation (including hardware in the loop), and fast prototyping of distributed WSN applications. The framework is based on tools widely used in industry like MATLAB Simulink® [21] and Stateflow® [22]. In addition, the architecture of the application is described using the standard Web Service Definition Language (WSDL).

38.3.1 Development Flow Overview

In the approach presented here, the Simulink® and Stateflow® graphical design tools are used for design entry using high-level abstract concurrent models, which simplify the design, simulation, and prototyping phases.

The abstract model can be automatically translated to simulation models that can be used on widely used network simulators such as OMNeT++/MiXiM [36]. The same model can also be translated for direct implementation on embedded operating systems, like TinyOS and Contiki, for hardware-in-the-loop simulation and deployment.

The framework shown in Fig. 38.2 provides support for target application design using high-level abstract models, without requiring knowledge of the low-level

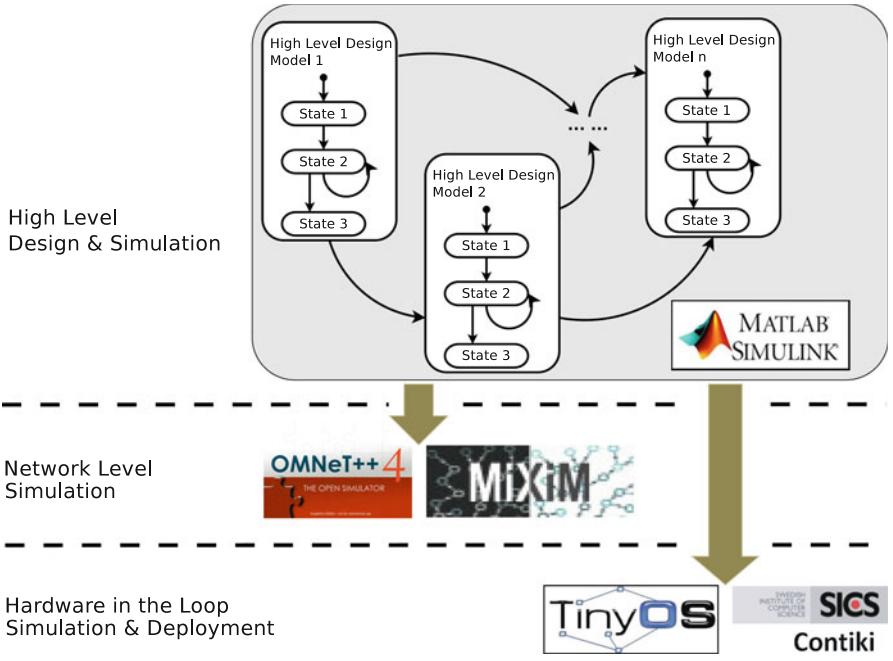


Fig. 38.2 Overview of the development flow based on Simulink® framework

specifications of the underlying hardware and software platforms or communication protocol stacks. It also allows one to automatically reuse the code generated from the same model for different simulation environments and deployment platforms.

Within the framework, the target application is first decomposed into a set of interconnected high-level object-oriented abstract models. These can exchange messages in a service-driven fashion. The model internal logic is described intuitively using a visual programming language based on Stateflow® StateCharts [20] and Simulink® block diagrams, as shown in Fig. 38.2.

38.3.2 Component Structure

The framework allows the designer to define the structure and the behavior of the WSN application by means of self-contained high-level abstract functional modules (Simulink® blocks) like the one shown in Fig. 38.3. Each module is seen as a "black box" by the other modules, being externally characterized by its tunable attributes and by the used and provided services.

Services used by the module are imported through its inbound service ports, such as InServ_1, ..., InServ_Y in Fig. 38.3. Services provided are exported through the outbound service ports of the module, such as OutServ_1, ..., OutServ_Z. A module can use also bidirectional service ports (e.g., InOutServ_1, ..., InOutServ_X) to connect an inbound service port and an outbound one to represent an instance of a combined service. This service is both used by the module and requires a response by its provider. Each service instance of a module (used, provided, or combined) is associated to an interface, which defines the contents of the messages transmitted by that service.

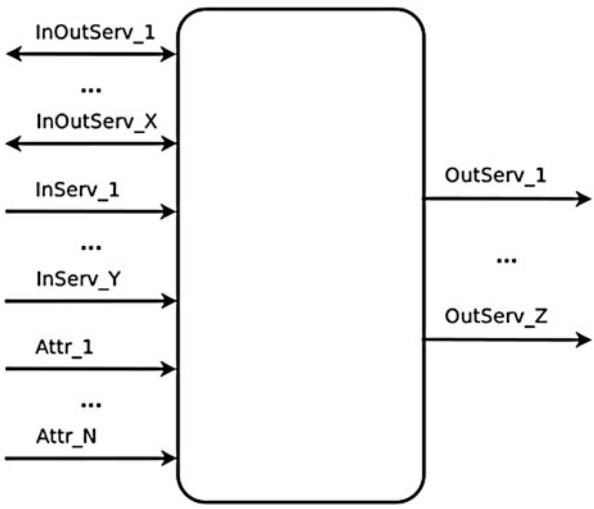


Fig. 38.3 Self-contained high-level abstract functional module (Simulink® block)

A module can also expose tunable attributes (e.g., Attr_1, …, Attr_N in Fig. 38.3) which are meant to allow the developer to adjust the performance of the module without significant changes to its internal logic. Like the services, attributes are associated to an interface that defines their constituents.

The WSN applications running on nodes can be modeled as an interconnected set of modules. Therefore, the internal details of the modules do not depend on the external entities and they can be loosely interconnected. Each Abstract Design Module (ADM) carries out part of the functions of the target application by exchanging service messages with the connected modules, through its service ports. An outbound service port of a module can be connected to any inbound service port of another module, as long as they share the same interface type. Inbound and outbound service ports can remain disconnected, which means that no incoming service messages are imported by the floating inbound service port and that the outgoing service messages on the floating outbound service port will be discarded (of course the designer must make sure that these missing connections do not impair the overall application functionality). These are similar to unidirectional function calls and can be used to transmit service-specific messages between modules without exposing the internal implementation details.

Module behavior is represented using an event-driven hierarchical Finite-State Machine (FSM) in the form of a StateChart, as shown in Fig. 38.4. The logic flow, i.e., the change of the active state, is determined by either its internal default transitions or by external service messages imported from other modules. These are processed by the FSM based on the values of its tunable attributes, and the computation results are attached to the appropriate outgoing service messages which are sent out through the corresponding outbound service ports.

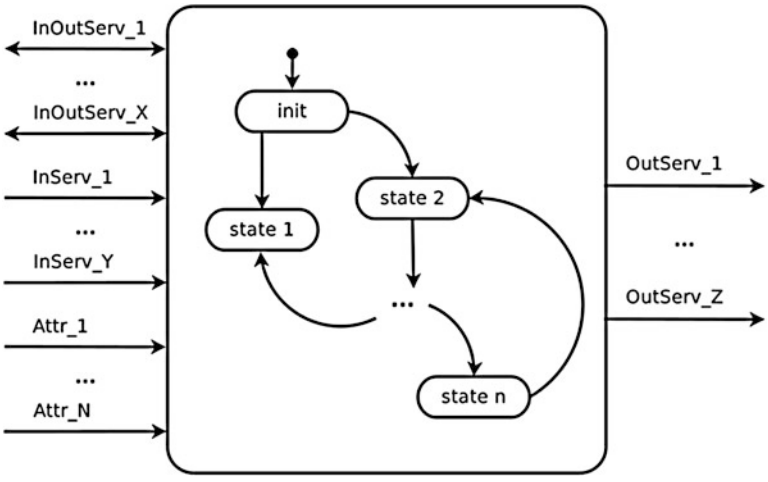


Fig. 38.4 A StateChart implements a finite state machine which defines the behavior of a functional module

User-defined operations can be attached to each state or to state transitions. They will be executed upon the entry, permanence, or the exit phases of each state.

The module has a different representation in the various steps of the workflow, serving different purposes. For instance, it can be viewed as a native Simulink®/ Stateflow® block for modeling and single-node functional simulation, as an OMNeT++/MiXiM module for large network simulations, or as a TinyOS component or Contiki OS process for deployment on the target WSN node.

38.3.3 Design Flow

The workflow illustrates the basic operation of the framework. It uses an iterative V-shape flow that starts with the requirement analysis, as shown in Fig. 38.5. It is made of three development task types:

- Manual tasks include development activities that are not directly supported by the framework and must be manually performed by the designers using other development tools.
- Supported tasks include some activities performed by the designers, with direct tool support from the framework.
- Automatic tasks are fully supported and performed by tools in the framework.

In the following, we describe the steps of the workflow in more detail.

38.3.3.1 Requirement Analysis

Requirement analysis is the first step in the workflow. It is a manual task and it consists in the analysis of the requirements of the target WSN application. It includes a list of the required functions and attributes supported by the application, such as:

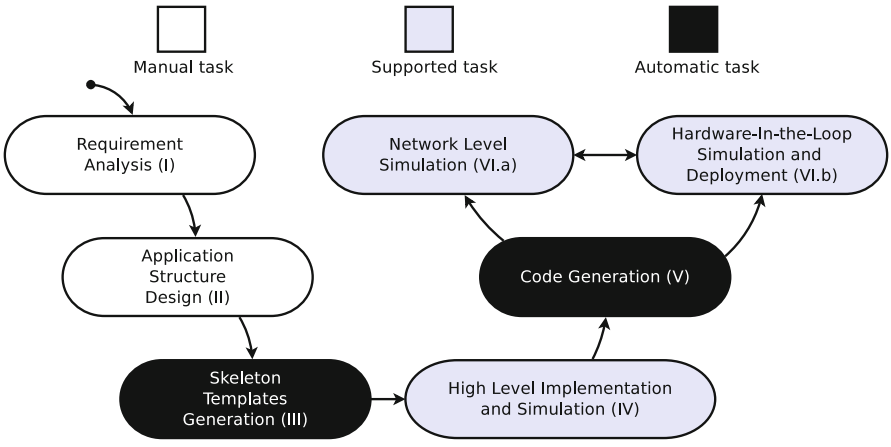


Fig. 38.5 Framework development flow is based on a V-shape iterative model

- what measurements will be performed by the node;
- what operations are expected from the nodes;
- what state variables (attributes) will be exposed as the tunable attributes by the application;
- what kind of criteria will be employed to validate the application and evaluate its performance.

When these requirements are defined, the designer can move to the next step, to describe the modules.

38.3.3.2 Module Description

Module description is based on the results of previous analysis. The developer can decompose the target application into a set of interconnected modules, each implementing a part of the target application functions by exchanging service messages with other connected modules through its service ports.

In this step, the developer lists the services and the attributes that are included in each module or includes them from a library of preexisting descriptions (e.g., defined in previous designs). For each module, the developer defines a description file with all services and tunable attributes of the module, such as the service name, interface, and type of associated service ports.

For instance, for an application that can be decomposed into a set of modules as shown in Fig. 38.6, the developer will define the content of the service messages exchanged among the modules and obtains a model description file for each module. Once defined, the description files are provided to the framework to automatically generate the skeleton templates for the modules in the next step.

38.3.3.3 Generation of an Application Skeleton

The skeleton template is defined in terms of Stateflow® blocks. In the generated skeleton template, all the services and attributes defined by the developer for that module in the previous step will be interpreted as a port. A combined service will be mapped to a pair of input and output ports in the skeleton template. For instance, the module shown in Fig. 38.4 is instantiated as shown in Fig. 38.6.

Each inbound service will be mapped to an input port associated with the specified data type defined by its interface (e.g., InServ_1 in Fig. 38.3 to InServ_1 in Fig. 38.7), each outbound service is mapped to an output port (e.g., OutServ_1 in Fig. 38.3 to OutServ_1 in Fig. 38.7), and a combined service is mapped to a pair of input and output ports (e.g., InOutServ_1 in Fig. 38.3 to InOutServ_1_IN and InOutServ_1_OUT in Fig. 38.7). For each input and output port, a corresponding driver function is automatically generated, such as those shown in Fig. 38.7:

- driver_InServ_1 for InServ_1
- driver_OutServ_1 for OutServ_1
- driver_InOutServ_1_IN and driver_InOutServ_1_OUT for InOutServ_1

These driver functions are used to detect the incoming service messages and to send out the outgoing service messages for the input ports and output ports,

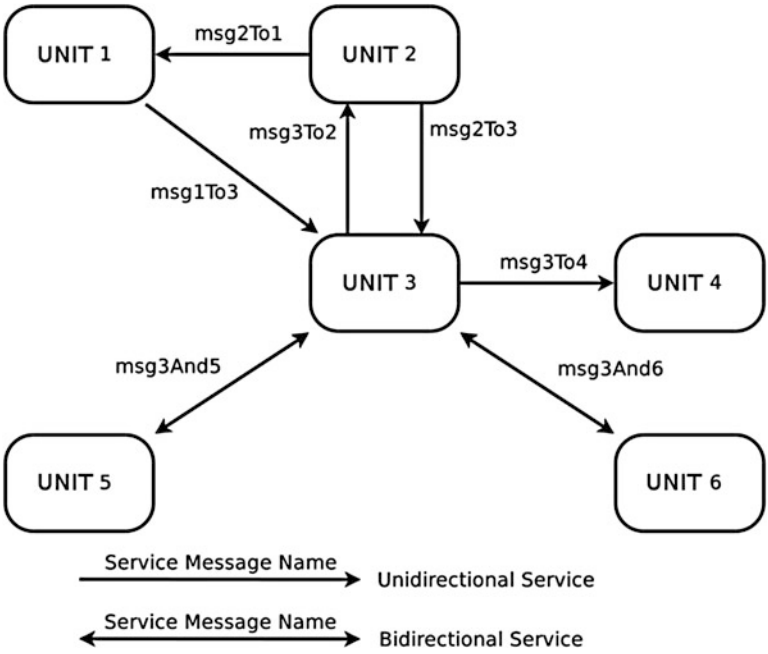


Fig. 38.6 Example of WSN node application decomposition in functional modules

respectively. Similar to the inbound services, for each tunable attribute, the framework will generate an input port and a port state variable (e.g., Attr_1 and var_Attr_1 in Fig. 38.7 for Attr_1 in Fig. 38.3), through which the developer can externally set the desired value for that attribute.

38.3.3.4 Customization of the Application Skeleton

The skeleton template can be customized through a supported task that includes two types of activities in the Simulink®/Stateflow® environment, namely, skeleton template completion, to create a full module, and module composition.

Skeleton template completion is done by the developers by modifying the automatically generated skeleton template with the desired internal functions. This consists in processing the imported service messages based on the values of the exposed tunable attributes and generating the corresponding outgoing service messages.

The internal logic of a module is defined by the developer using StateCharts and block diagrams, without knowing the details of the target platform (WSN node).

As shown in Fig. 38.7, the developer-defined operations implemented within the states are executed upon entry, permanence, or exit phases of each state, while the developer-defined operations implemented between two connected states are executed when the state transfer occurs through that state connection. All these developer-defined operations can execute any developer-defined local functions

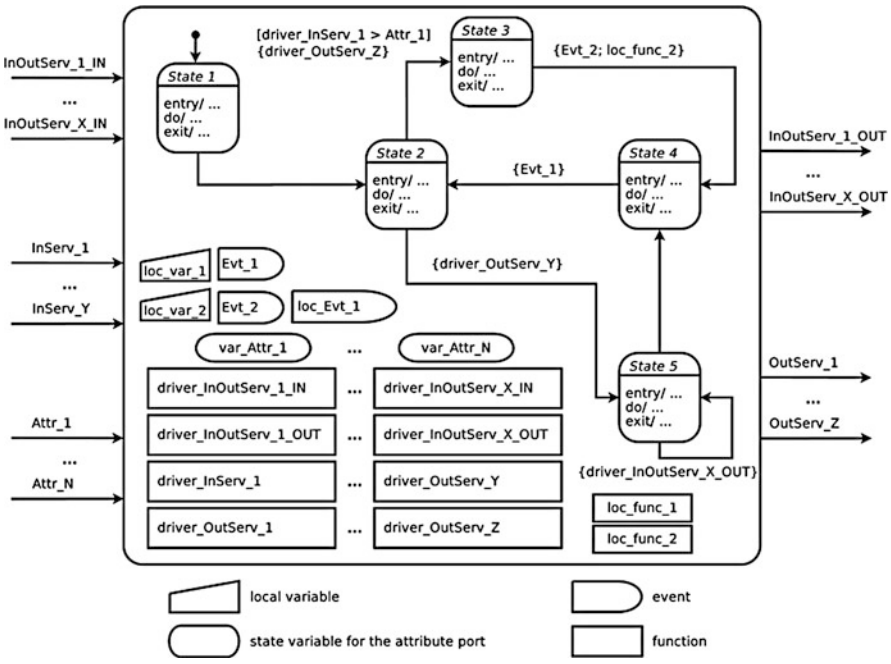


Fig. 38.7 Example of complete WSN node application skeleton

(e.g., loc_func_1 and loc_func_2 in Fig. 38.7) to perform computational tasks, as well as generate outgoing service messages. Besides the externally tunable attributes, additional local variables (e.g., loc_var_1 and loc_var_2 in Fig. 38.7) and local events (e.g., loc_Evt_1 in Fig. 38.7) can be freely defined within each module.

If a single FSM is not sufficient to model the desired function, a single Simulink® block can contain one or more sub-charts which can be integrated into the main FSM in either sequential or parallel execution order, sharing the set of incoming and outgoing messages, attributes, local variables, and local events.

The module composition phase creates the final application. This is done by wiring each outgoing port of each module to the relevant incoming port(s) of other modules and by assigning proper values to the attribute port(s) of each module.

Then the high-level application model can be used in the next step to automatically generate an implementation for different platforms.

38.3.3.5 Code Generation and Deployment

Since the framework helps the developers to automatically port the same high-level design to different platforms, the developers can easily refine their design and explore the hardware/software trade-off space almost without low-level detail knowledge. This is an important benefit, since design porting to a different platform is often effort-intensive and error-prone.

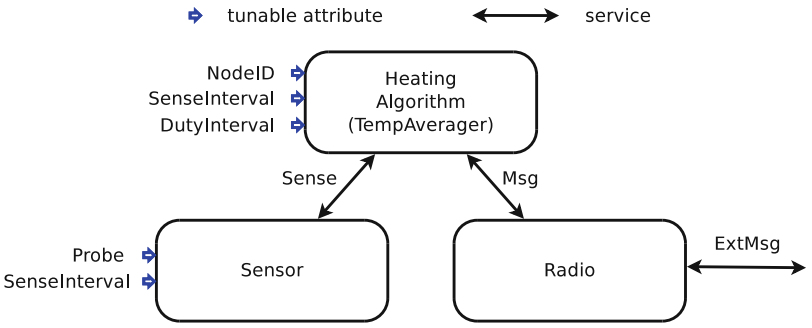


Fig. 38.8 Module structure of example node-level WSN application

We will analyze a use case in which the nodes collect and perform a distributed processing of the data from a temperature sensor. Each WSN node wakes up periodically to carry out the following tasks:

1. sample the temperature values with the desired sampling frequency;
2. collaboratively average these values with those from its one-hop neighbors within a sliding time window;
3. broadcast the calculated average temperature value to the neighbors.

The requirement analysis is used to drive the application architectural design phase, where the developer decomposes the application into a set of interconnected ADMs, each carrying out a part of the entire functionality. Since the ADMs are characterized by services and attributes exposed on their boundary, their internal behavior may not be detailed in this step. The developer just assigns the requirements listed earlier to the constituent ADMs by defining their boundaries. This can be done describing the ADM services and attributes either manually or by importing them from a library (e.g., created in previous designs).

In the proposed use case, the following parameters have been identified as potential tunable attributes for each node:

1. its own node identifier (NodeID);
2. sample refresh interval inside the averaging algorithm (SenseInterval);
3. the sampling period of the temperature sensor (SamplingInterval);
4. the size of the time window to compute averages, which is equal to the node duty interval (DutyInterval).

Based on requirement analysis (first step in Fig. 38.5), the design is split in three interconnected ADMs: a `Sensor` module, a `Radio` module, and an `Algorithm` (`TempAverager`) module, as shown in Fig. 38.8. The Sensor module samples and preprocesses temperature data. The Radio module interfaces with the protocol stack for short-range communication with neighbor nodes. The TempAverager module

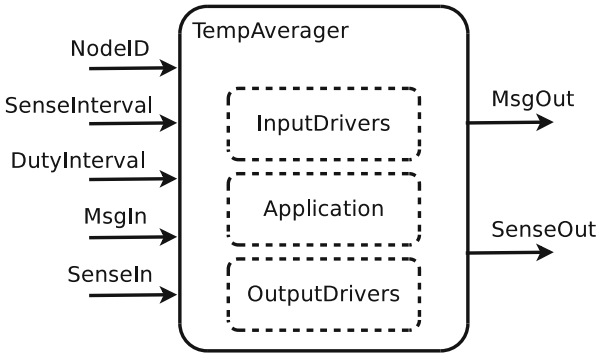


Fig. 38.9 Overview of the generated skeleton template

handles all onboard data processing. Both the Sensor and the Radio modules are connected to the TempAverager module to exchange service messages (e.g., `Sense` and `Msg` in Fig. 38.8).

The developer manually defines a boundary description file for each ADM. These are imported in the framework for the next step, template generation.

ADM description files are then supplied to the framework which maps them automatically to skeleton templates that are defined as Stateflow® blocks. All ADM services and attributes defined in the ADM templates are interpreted as ports or port pairs for a request-response service.

For each port, a driver function is automatically created inside the skeleton template, which hides the low-level Simulink® handling of signals and service messages. These functions handle the service messages exchanges through ports. Like the input services, each tunable attribute has an input port and a state variable allowing to set the attribute value from outside the ADM.

Figure 38.9 shows the skeleton template generated automatically for Stateflow®. Each skeleton template is created with three FSMs that run in parallel: `InputDrivers`, `Application`, and `OutputDrivers` which can be used to implement the application logic.

Once the skeleton template has been filled with functional details, simulated, and debugged, it can be used for automatic code generation. A framework tool converts the high-level and platform-independent design into target code that runs in different network simulation environments or on different target OSs and platforms. These can be:

- Simulink®/Stateflow® can be used for node-level and small-scale network simulation;
- OMNeT++/MiXiM can be used for large-scale network simulation. Each ADM in the WSN application is mapped to a component, which is the programming unit used by all these simulators;

Table 38.1 Code size and the memory usage for the use case application implemented on top of TinyOS using a Telos B node

	ROM [bytes]	RAM [bytes]
Handwritten	17,220	492
Framework-generated	20,562	526

- TinyOS [17] can be used for code deployment on target nodes. Each ADM of the application is automatically converted to a TinyOS module written in nesC [13] containing the ADM internal logic;
- Contiki OS [10] can also be used for deployment. In that case, each ADM is instantiated as a protothread. The generated code can also be run in the COOJA simulator.

For instance, we used the framework code generation function to convert the high-level design ADMs to nesC modules that are suitable for a simple network composed of Memsic Telos rev. B nodes running TinyOS. The generated nesC modules (`Radio`, `Sensor`, and `TempAverager`) are configured, interconnected, and encapsulated in a wrapper nesC module that is then wired in TinyOS to use the existing radio communication services.

Table 38.1 shows the code size and memory usage measured for the binary code generated using the development framework and the same application logic implemented manually. The results show a penalty for the generated code of less than 20% in terms of code size and less than 7% in terms of data Random-Access Memory (RAM) requirements.

38.4 Automated WSN Application Composition

The MBD tool presented in Sect. 38.3 requires the designer to explicitly compose the modules to implement a full application, which is an effort-intensive process.

A different set of tools can automate the application composition phase starting from a high-level application specification and an existing library of reusable modules. The toolset can further speed up WSN application development and the exploration of the design space, as will be discussed next.

38.4.1 Development Flow Using Automated Application Composition

Figure 38.10 compares the automated design flow (shown in the lower part) with a typical node-level WSN application development flow (shown in the upper part). The automated flow accepts a high-level application-centric system description at node level and can be integrated with various external tools, each of them used to assist the developer in specific tasks.

38.4.1.1 Development Flow Overview

The automated flow [2] starts with the input of the application-specific behavior encapsulated in a component format (described later in Sect. 38.4.1.4).

The top-level component and all library components have the same format, with two major sections: a code section and a metadata section. In the first step of the flow in Fig. 38.10, the designer fills both of them for the top-level component, as follows.

The code section can store different types of code (behavioral, simulation models, etc.). These are always considered as (possibly parameterized) black boxes by the system synthesis engine; thus, there are no restrictions on the coding language or the representation format (which can be also binary code for one or more

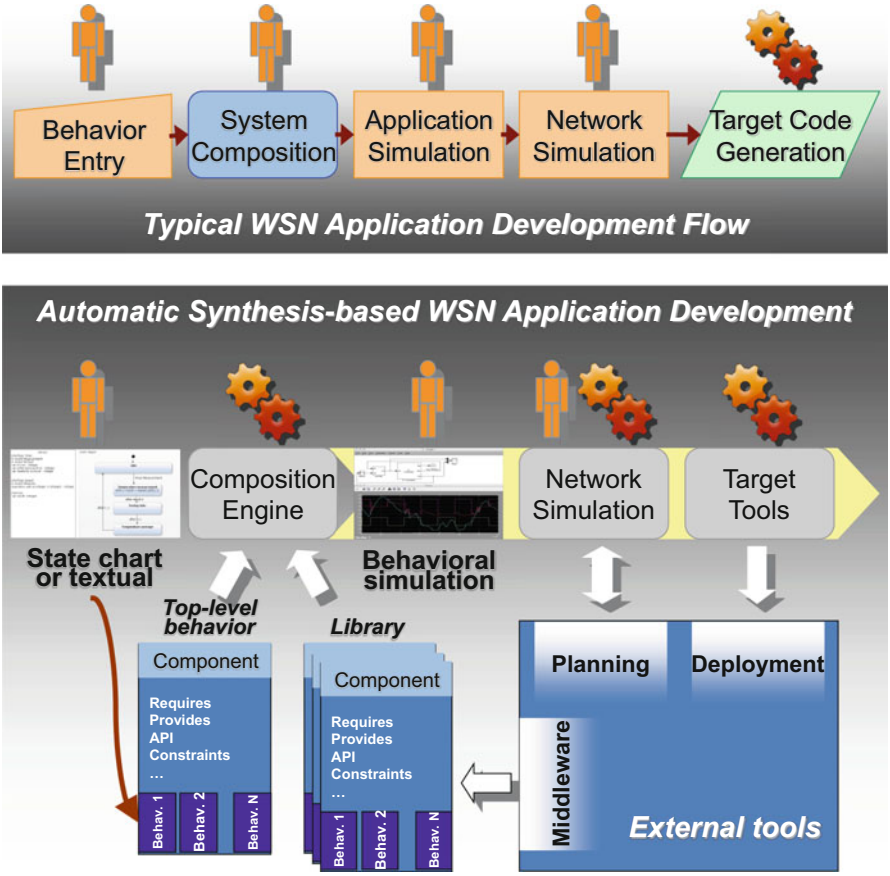


Fig. 38.10 Comparison of the main stages of manual (*top*) and automated (*lower part*) node-level WSN application design flows. A human body tags manual phases while a gear tags automatic ones. The automated flow accelerates mainly the system composition and the preparation of the network simulation

target platforms). Hence, the behavioral code of the component can come from various sources, ranging from manually written source code (e.g., legacy C or nesC code) to code generated by high-level development flows (e.g., metaprogramming approaches [24], Unified Modeling Language (UML)-based or ad hoc high-level modeling flows [8, 29, 32], ▸ Chap. 5, "Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design Approach"). MBD tools can also generate suitable behavioral code, for example, the Stateflow® tools used in Sect. 38.3 or the Yakindu Statechart Tools [26] (both provide a state chart-based integrated modeling environment for the specification and development of event-driven systems).

The metadata section of the components is used by the subsequent phase of the flow in Fig. 38.10, namely, the automated system composition. This phase uses only the semantics of the metadata to automatically select the components (more precisely, through parameter-based selection and customization) to compose the node system, both as hardware and software. If the designer uses the flow described in Sect. 38.3, then the component metadata are generated automatically from the module description. Otherwise, the designer should manually enter the metadata.

Figure 38.11 shows one possible integration of the automatic flow with external WSN development tools. The flow shown on the left side of the figure supports WSN network planning using specific tools for input of geographical data (e.g., of a topographic map), selection of node locations in the field, and Radio Frequency (RF) propagation simulation to estimate node connectivity.

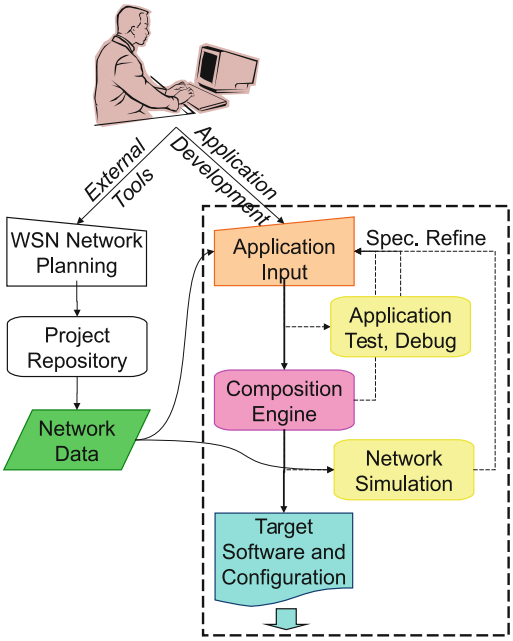


Fig. 38.11 The main phases of the application development flow based on automated system composition and one possible interaction with external tools. The application developer describes both the network layout and composition using external tools (flow on the *left*) as well as the node-level application (flow on the *right*). Where necessary, the application development flow can extract from network description the distinct types of nodes and the network connectivity information. The former is used to create projects for node-level application development, while the latter is needed to prepare the network simulations

The application development flow shown on the right side of Fig. 38.11 can retrieve the network planning data from the project repository. It uses it to extract the number of distinct node types in the network and to create a skeleton project for node-level application development for each type.

For each project, the developer inputs the application in the format of a top-level component, as we mentioned earlier. When the developers use model-based design flows for application input, such as Stateflow® or Yakindu SCT, they can use the features of these design environments to test and refine the application at this high level of abstraction.

The next step, system composition, is fully automated by the composition engine. The engine starts by processing all metadata of the top-level component, such as requires, provides, and conflicts. These encode what is needed by the component in order to operate properly in terms of hardware, interfaces, configurations, etc. (requires), what the component can provide to satisfy the requires of other components (provides), and in which conditions the component cannot operate at all (conflicts).

These properties drive the composition process, which iteratively looks for all subsets of the library of components that do not have any unsatisfied requirements left and, at the same time, satisfy all constraints imposed by the top-level and the other selected components. Each such subset represents a possible system solution that satisfies application specifications. These solutions are automatically saved and can be further examined or manually modified by the developer or used as they are.

For each generated solution, the composition tool can create simulation projects, as shown in the next steps of the flow. The simulations are set up to run on external simulators (e.g., OMNeT++ [36]) and can be at various levels of abstraction. Basically, this consists of the extraction and configuration of the suitable simulation views from the components of the solution and their assembly in simulation projects.

Using a similar mechanism, the composition engine generates the projects that can be compiled with the target tools to create the programs for the WSN nodes. These projects are typically generated in the format expected by the target tools, which often is a make-based project.

Moreover, the components that are instantiated in the solution can include a bill of materials (e.g., compatible hardware nodes, RF and transducer characteristics) or software dependencies on specific compilation toolchains or underlying OS. The composition engine can collect all these, e.g., into a solution-specific Bill of Materials (BOM) and compilation requirements.

As shown in Fig. 38.11, after each step, the developer can analyze the results and attempt to optimize them either by changing the specification (and rerunning the composition) or by manually editing the generated projects.

As mentioned, the benefits of WSN application automated composition are compounded by its integration with external tools, such as simulators, target compilation chains that can provide inputs or assist the developer in other phases of the flow. For instance, Fig. 38.10 shows some typical interfaces with middleware [23, 25], WSN planning tools [30], or deployment and maintenance tools [18]. An example of integration is presented in [2].

However, the wide variety of the existing tools and models makes it very difficult to define an exhaustive set of toolset external interfaces. Moreover, rigid toolset interfaces or operation models can reduce its value and hamper its adoption in the rapidly evolving WSN context, which does not seem to be slowed down by standardization efforts or proprietary Application Programming Interface (API) proposals. Thus, as we will show later on, an optimal tool integration in existing and future development flows would base its core operation on a model expressive enough to encode both high-level abstractions and low-level details. Moreover, it is also important to provide well-defined interfaces and semantics to simplify its maintenance, updates, integration with other tools, and extensions to other application domains.

38.4.1.2 Automated Composition Tool Overview

The main functions of the tool are application input (interface and processing), automated hardware-software composition, and code and configuration generation.

Application domain experts can benefit most from an interactive user-friendly interface for the description of the WSN application top-level behavior. Stateflow®, as described in Sect. 38.3, are well established in this regard for their intuitive use, and they can also provide suitable high-level models to facilitate the description of the desired application domain behavior. On the other hand, the tool can accept application descriptions generated by other tools, such as middleware [12] or metaprogramming [24].

Automated composition of hardware-software systems able to support WSN application specification shields the developer from most time-consuming and error-prone implementation details. At the same time, the composition increases the reuse of functional components from the library, which can be software components (e.g., OS, functional blocks, software configurations, project build setup), hardware components (such as WSN nodes, transducers, radio types or specific devices, hardware configurations), and specifications (e.g., target compilation toolchain, RF requirements).

While the tool performs some consistency and satisfiability checks of application specifications in order to reject early those that cannot have a solution, other incomplete specifications are accepted because the tool can typically infer default parameters based on the values provided by the library components and heuristics. This allows the developer to refine the specifications during successive design iterations using also the results of previous underspecified composition runs.

Incomplete specifications may lead to the composition of incomplete systems, which nevertheless satisfy every requirement. This can save effort for experienced developers, who can use the resulting incomplete projects as starting points for manual refinements.

Code generation can produce simulation or target compilation projects. Network simulations can be configured using the simulation models of the components of the solutions, their parameters, and the actual configurations. Realistic communication channels defined by a planning tool [30] can be used, if available. In a similar way,

the tool uses the implementation code of the components instantiated in a solution to generate and configure the project that compiles the code for the WSN nodes.

Besides this highly automated process, the tool allows the experienced developers to take over manually the application development at any stage: design entry, testing and debug, system composition, node application simulation, network simulation, and target code generation. Basically, this is achieved by:

- making use of textual data formats that can be edited with general purpose or specialized editors;
- documenting the data formats, their semantics, and processing during each phase of the development flow;
- including well-known tools in the flow with clean and well-documented interfaces to simplify their update or replacement for the specialization of the flow;
- allowing one to run manually the individual tools, even outside the integrated flow, e.g., to explore options and operation modes that are not supported by the integrated flow.

38.4.1.3 Automated Composition Tool Input Interface

As argued above, abstract concurrent Stateflow® are an intuitive and efficient high-level means to specify the top-level application behavior. Besides the behavior, the tool should support the specification of interfaces and other requirements of the behavior. These are necessary because the flow does not make any assumptions about the format, the language, or the modeling of the behavioral part.

All these data are captured in the top-level component of the design that is then used to drive the system composition engine. Using library components, the engine attempts to automatically compose a hardware and software system that supports the application-specific behavior and provides all its requirements.

For instance, let us consider a WSN application that collects and sends every five minutes the environmental temperature during four intervals of two hours spread evenly during the day. The functional description of this application consists of a periodic check if the temperature collection is enabled. If it is enabled, then it checks if five minutes have elapsed from previous reading, and if so then it acquires a new reading and sends it to the communication channel. The whole application behavior can be encoded in just a few condition checks and data transfers, plus some configuration requirements to support them (such as timers, a temperature reading channel, a communication channel). The rest of the node application and communications are not application-specific; hence, the developer should not spend effort developing or interfacing with them. In this flow (see Figs. 38.10 and 38.11), these tasks are automatically handled by the composition engine, which attempts to build a system that satisfies all specifications by reusing library components, as will be explained later.

The top-level component can include also several types of metadata properties. For instance, if the IPv6 over Low Power Wireless Personal Area Network (6LoWPAN) protocol is a specification of the WSN application, a requirement for 6LoWPAN can be added to the top-level component, regardless if the

top-level component functional code interfaces directly with the field communication protocol. This way, the 6LoWPAN requirement directs the application composition to instantiate the functional components from the library that provide this communication protocol. However, the tool will instantiate only those 6LoWPAN components that satisfy other system requirements that are collected from both the top-level and other instantiated components.

38.4.1.4 Structure of Top-Level and Library Components

Library components are central to the operation of the system composition engine (see Fig. 38.12). They are used for:

- the definition by the developer of the behavior and requirements of the node-level WSN application, modeled as a top-level component;
- the definition of library blocks that can be instantiated by the composition tool to compose a hardware-software system that satisfies all design specifications;
- the interface with OS or middleware services when necessary, to support the functionality of the application;
- providing the simulation models, at different levels of abstraction;
- providing the target code that is used to build the projects as well as to configure and compile the code for the target nodes;
- providing code generators that can be run by the composition tool to either:
  - check if the component can be configured to satisfy the requirements derived for the current partial solution during composition, so that it can be instantiated in the solution;
  - build specialized code stubs, e.g., for API translation and component code configuration, that are based on the actual parameters of the solution in which they are instantiated;
- providing hardware component specifications, which are collected in a BOM;

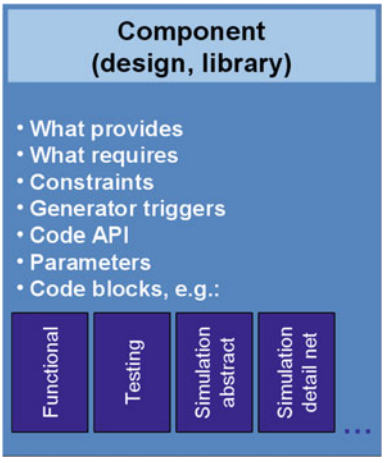


Fig. 38.12 *Top*-level application specification component and library components share the same structure: a variable set of views (shown darker on the *bottom*) that are handed as *black boxes* by the system composition process and a set of metadata that express the requirements and the capabilities of the component. The components are encoded in XML (EMF XMI)

- providing nonfunctional requirements, such as for special radio-frequency requirements or compilation toolchains.

Yakindu SCT was used in this specific case to generate and modify the library components, including their metadata. Hence, the components are encoded using extensions of Yakindu projects, which use the Eclipse Modeling-Framework (EMF) XML Metadata Interchange (XMI) format [27]. XMI is an XML interchange format well supported especially by UML-based tools. Components in other formats can be supported using suitable translators, as long as those formats can adequately represent the meanings of the metadata and the functional models of the Yakindu components.

The library components are designed to be compatible with the concurrency and communication models provided by the underlying OS or middleware abstractions. To achieve a consistent system composition, all external communications among and with the components need to go through their exposed interfaces in order to be visible to the system composition engine.

38.4.1.5 System Composition Process

To exemplify the composition process, we show in Fig. 38.13 a simplified representation of just a few metadata properties for both the library components (bottom) and the top-level specification component (top).

At the begin of the system composition process, the system composition engine is driven by the metadata specifications of the top-level component of the design, and its selections are guided by the metadata of the components in the toolset library. As system composition progresses by instantiating library components in the partial solution, the metadata of the instantiated components will drive the search performed by the engine alongside with the still unsatisfied specifications of the top-level component. During the entire composition process, the top-level component and its metadata are considered mandatory. However, the library components can be instantiated and removed from the solution as necessary, to satisfy the design requirements.

More specifically, at the begin of the system composition process, the engine loads all metadata from the top-level component and all library components. Then the recursive solver of the engine starts to build a partial solution by looking for library components that match the requirements of the top-level specification component. It instantiates these components, one at a time, into the partial solution, and then it repeats the process. This time, it considers the specifications of all the components that are currently instantiated in the partial solution, including the top-level component.

The solution becomes complete when all the requirements of its components are satisfied. Once a complete solution is found, it is saved along with the actual values for all its configuration parameters. Then the solver resumes the search for other solutions by removing components from the current solution and replacing them with alternatives, if any. The solver basically stops when all possible combinations have been tried. As a future development, the composition engine can be coupled

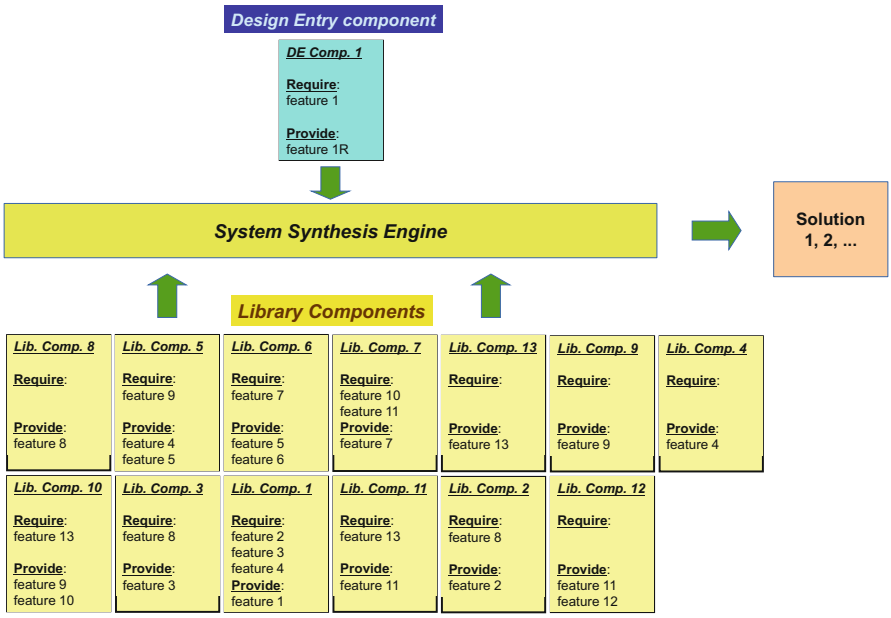


Fig. 38.13 Simplified example of metadata for the design specification component and some library components

with design space exploration tools, e.g., [28], ▸ Chap. 6, "Optimization Strategies in Design Space Exploration" and ▸ Chap. 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration".

For example, considering the top-level specification and the library components shown in Fig. 38.13, the system composition engine loads first the design entry top-level component and all 13 components of the library. Then the engine explores all component combinations that can lead to a complete solution, i.e., a component composition where all mandatory component requirements are satisfied. All complete solutions found are saved. For the example shown in Fig. 38.13, these are:

1. solution using components 1, 2, 3, 4, 8;
2. solution using components 1, 2, 3, 5, 8, 9;
3. solution using components 1, 2, 3, 5, 8, 10, 13.

The format of the saved solutions includes all the elements necessary to instantiate, connect, and configure the selected library components. By revisiting these data and the instantiated components, the engine is able to:

- extract some figures of merit for the solutions from the instantiated components metadata and their actual configuration within the solution, e.g., FLASH and RAM requirements, communication protocol characteristics, etc.;

- generate the BOM and nonfunctional specifications, such as what are the compatible compilation chains;
- generate and configure network simulation models;
- generate make-based projects that can build the programming and configuration code for the target nodes.

The developers can use these data to decide which solution, if any, is suitable. Alternatively, they may decide to change the application specifications in order to improve the solutions or to manually optimize a promising solution.

38.5 Case Studies

In the following, we will present the application of the automated composition flow to two different representative WSN applications of practical interest, which are described in [2].

One application is a self-powered WSN gateway designed for long-term event-based environmental monitoring. It can handle up to 1000 sensor nodes, process, and aggregate their messages, bidirectionally communicate with a server over the Internet using Transmission Control Protocol/Internet Protocol (TCP/IP) through a General Packet Radio Service (GPRS) modem, and receive remote updates. Its hardware requirements are very small, comparable to those commonly used to implement a WSN sensor node. To satisfy these requirements, the original gateway code was handwritten fully in C, without using an embedded OS or external libraries besides the low-level standard C libraries.

The other application is a typical WSN sensor node for remote environmental monitoring. It has transducers for some pollutant gases and it is designed to operate near industrial sites adjacent to urban areas. The node was developed on top of ChibiOS [33], a real-time, preemptive, small, and fast embedded OS.

In both applications, we have started from an existing implementation. However, the flow presented below can be used both for porting legacy code on the toolset and adding toolset support for new hardware:

Create library components. The system composition engine is designed to make extensive use of the components in the library. Hence, the quality of its library strongly determines the quality of the composed systems.

A good quality library should include enough variety of building blocks to support most sensing requirements (e.g., various types of sensor interfaces), processing requirements (e.g., queues, stats, encryption), in-field and out-of-field communication protocols, etc.

The libraries can and should be reused for several designs. Thus, once a library is created, it may receive incremental updates (e.g., support for new sensors or new algorithms) or significant additions (e.g., support for new hardware nodes or embedded OSs).

Create the top-level specification component. This component is application-specific and drives the whole system synthesis process. It needs to include enough requirements to cover all application needs without being over-specified, which would restrict too much the search space of the synthesis engine.

Run the system composition engine. The engine attempts to solve all requirements of the top-level component using the existing components from the library.

Evaluate the solutions. As shown in Sect. 38.4.1.5, the toolset can extract and calculate various figures of merit for each solution which can be used by the developer in order to select a suitable solution. Moreover, the solutions can also be manually analyzed and further tuned.

The applications that we consider here are based on existing projects. In these cases, particular attention should be given to the creation of the components from the existing hardware or software Intellectual Property (IP) blocks in order to allow the toolset to find at least a solution that matches the existing projects. One obvious and easy-to-automate way is to pack the IP code in an appropriate component model (see Fig. 38.12). Then, for each component, it is important to properly describe its functional elements, such as its interfaces and configuration capabilities, and the semantics associated to component behavior and data exchanges.

38.5.1 Full-Custom WSN Gateway

The original gateway project was implemented with limited hardware resources, which are typical for WSN sensor nodes. It included an AVR ATmega1281 microcontroller, two CC1101 radio modems operating in the 433 MHz band using a proprietary communication protocol. These connected the gateway, on separate channels, both with the peer gateways and with much smaller sensor nodes, which were used for high-density environmental monitoring. The gateway included also a GPRS TCP/IP-enabled modem for long-range communication with the server and for remote updates.

The application software of the node is written entirely in C, without an embedded OS. It is made of 49 modules, each of them implementing a well-defined function: generic functions that are used by most applications (like the task scheduler, oscillator calibration, or the message queue) or specialized functions that are used for specific applications (such as drivers for specific onboard sensors).

Instead of an embedded OS, the code uses a module that implements a round-robin scheduler that can periodically run statically assigned tasks. Most tasks are implemented as FSMs using the coroutine approach. Each task executes for a minimum amount of time when started and voluntarily yields the processor whenever it completes its processing or it needs to wait for some reason. Also, each task is responsible for maintaining its own state and persistent data between calls, in order to be able to resume its execution upon its next scheduling slot.

Besides the functional blocks needed to implement the main gateway behavior, the code has several modules that implement safety and error recovery functions.

Also, there are several driver and processing modules for several sensors and auxiliary devices that can be mounted directly onboard the gateway node:

`adc` Drivers for the Analog-to-Digital Converter (ADC) peripherals.
The module captures the ADC interrupt and calls the conversion data processing function.
`anemometer` Weather anemometer sensor handling functions.
Driver and controller for the anemometer transducer.
`battery` Utilities for battery reading processing.
The module provides the battery-specific voltage-to-capacity conversion tables and the functions to perform the conversion.
`cc` Field and mesh radio drivers.
The module handles everything related to the field and mesh radio onboard the gateway.
`crc` Cyclic Redundancy Check (CRC) utilities.
Processing utilities (CRC calculation).
`eeprom` Electrically Erasable Programmable Read-Only Memory (EEPROM) driver.
EEPROM data structure and low-level I/O drivers.
`eeprom_ext` External EEPROM driver.
Driver for external EEPROM module.
`fc10` FC10 sensor handling functions.
It has both the top-level application and drivers for the FC10 transducer.
`field` Communication protocol with sensor nodes.
Processing of messages received from sensor nodes.
`geophone` Geophone sensor driver.
It has both the top-level application and drivers for the geophone transducer.
`gw` Node status.
Controls the state and configuration of the node.
`hal` Hardware high-level interface.
It processes asynchronous events from the network and onboard switches.
`humidity` Weather humidity sensor handling functions.
Driver and controller for the humidity transducer.
`hygrometer` Hygrometer sensor.
It has both the top-level application and drivers for the transducer.
`igwc` Internode communication.
Internode messaging and network formation.
`inst` Node installation mode.
Top-level application that runs during the installation of the node in the field.
`mesh` internode communication protocol.
Processing of node-level messages.
`modem` GPRS modem driver.
Driver for the GPRS modem.
`msg_filter` Messages queue filter.
Configurable application-specific processing of the queued messages.
`obs` Onboard sensor driver.
Drivers for various onboard sensors (not application-specific).
`oc_link` Operating center communication controller.
Controller of the connection with the server and server message preprocessor.
`power` Power module driver.
Driver for the power module.
`pressure` Weather pressure sensor handling functions.
Driver and controller for the pressure transducer.
`queue` Message queue.
Storage and processing of the messages queued to be delivered to the server.
`rain` Weather rain sensor handling functions.
Driver and controller for the rain transducer.

`rccal` Main Resistor-Capacitor (RC) oscillator calibration.
Performs the calibration of the internal RC oscillator.

`rel_mesh` Multihop message queue transfer via mesh, with acknowledge.
Bidirectional internode communication protocol.

`rpc` Remote procedure call.
Processors for remotely setting and querying (monitoring) node data and for sending remote commands.

`run_state` Execution health controller.
Module to monitor the state of the current run, i.e., how far the software execution has progressed since the last boot.

`sched` Task scheduler.
Scheduler.

`sensor` Sensor state and data processing.
Maintains the state of the sensors in range based on the contents of their messages (or lack thereof).

`sensor_ppc` Path passage counter sensor.
It has both the top-level application and drivers for a passage detector.

`service` Internal service requests handler.
Implements a service request/dispatch controller that can change the state of the node

`sio` Serial link for operating central message transfer.
Communication with the server over a wired serial line.

`spi` Master Serial Peripheral Interface (SPI) driver.
Driver for the SPI port.

`sr` Save-restore of RAM contents across watchdog resets.
Saves the contents of specific RAM areas before a watchdog reset and restores them after the reboot.

`sw` External switch driver.
Driver for the on-node switches.

`testing` Node testbench mode.
Various top-level applications that act as node and sensor node tester.

`test_tx_hw` Sensor testbench mode.
Top-level application for the node in testing mode.

`theft` Node antitheft detector.
Process that detects possible node theft actions.

`timer` Timer handler.
Provides several timers for use within the node.

`twi` Two Wire Interface (TWI) interface.
Driver for the TWI interface.

`usart` Universal Synchronous/Asynchronous Receiver/Transmitter (USART) drivers.
Drivers for the node USART ports.

`util` Utilities.
Various processing functions (e.g., conversion of bin values to American Standard Code for Information Interchange (ASCII) hex).

`version` Firmware version utilities.
It provides the version of node software.

`wd` Watchdog driver.
Driver for various functions attached to the watchdog timer.

`weather` Weather station handlers.
Top-level application that implements a weather station.

`zlist` RAM-efficient mapping of the Identifiers (IDs) of the sensor nodes in range.
Optimized storage and processing of the messages queued to be delivered to the server.

Most of these modules are made of several functions and may include sizable amounts of data. For example, module *queue* includes the data structures for buffering the messages in queues by priority, waiting to be transmitted, and the functions for the operation of the queues (such as query, addition, or removal). Similarly, the *sensor* module maintains the data structure with the status of all sensor nodes in range and provides the functions for their management, such as query or update.

Figure 38.14 shows the metadata of the library component that was generated for a very simple module, *version*. The module implements the function to store the gateway version information and provides methods to access it.

```
<sgraph:Gss xmi:id="_b2b65395f30689ed09f02e">
    <properties>
        <name>version_component</name><description />
    </properties>
    <views xmi:id="_08f5c2612c510ac5e105e7">
        <behavior>
            <view xmi:id="_5c39ae70c147735f28ad4b" name="version.c"
                type="source" language="C" encoding="base64">
                <description></description>
                <mem>LyoqCiAqIEBmaWxlIHZ [...]</mem>
            </view>
            <view xmi:id="_44e6770ca6e62fc2db54e9" name="version.h"
                type="source" language="C" encoding="base64">
                <description></description>
                <mem>LyoqCiAqIEBmaWxlIHZlc [...]</mem>
            </view>
        </behavior>
    </views>
    <resources>
        <behavior>
            <require><name>avr_libc</name><description /></require>
            <provide><name>version_component</name>
                <description /></provide>
        </behavior>
    </resources>
    <interfaces>
        <behavior>
            <provide>
                <description />
                <function>
                    <name>version_get</name>
                    <return><type>char *</type></return>
                    <port><ord>1</ord><type>char *</type></port>
                </function>
            </provide>
        </behavior>
    </interfaces>
</sgraph:Gss>
```

Fig. 38.14 Example of a simple library component that includes properties and a code view

At the top level, we can see the categories *properties*, *views*, *resources*, and *interfaces*. This simple component has only one property that contains the name of the module. The list of behavioral views includes two files corresponding to the source code of the module. The resources include one nonfunctional requirement to track the dependency of the component on a toolchain that supports the C functions used in the source code and a symbolic resource provided by the component which can be used, for instance, to directly require this component in design specifications or in other components. In terms of interfaces, the component provides a behavioral function, which retrieves and returns the version data. Additionally, for most metadata properties, one can enter a description that can be used, for instance, to help the developer understand the semantics of the component, when it is displayed in a component or solution editor.

Figure 38.15 shows the result of the composition of a minimal gateway system for which the specification was just to include the core gateway functions. Moreover, the composition tool ran the configuration helpers of the components, to set up their instances according to the actual values of their parameters, as found by the solver. For instance, the scheduler is automatically configured to support the actual tasks.

For this minimal requirement, the solver found a suitable composition with a maximum recursion depth of 888, matching 230 abstract requirements, 472 functional requirements, and two data requirements in less than 0.8 s on an 1.8 GHz Intel® Core™ i7-2677M processor.

In addition to software solution composition, the tool collects other requirements of the instantiated components into a BOM list that includes the hardware node type, radio specifications, and the target compilation toolchain.

By changing just the top-level specification component, we used the toolset to automatically compose systems for different application requirements (for different gateway compositions in this application).

adc	hygrometer	**rccal**	**test_tx**
anemometer	**igwc**	rel_mesh	**theft**
battery	**inst**	**rpc**	**timer**
cc	**main**	**run_state**	**twi**
crc	**mesh**	**sched**	**usart**
eeprom	**modem**	**sensor**	**util**
eeprom_ext	**msg_filter**	sensor_ppc	**version**
fc10	**obs**	**service**	**wd**
field	**oc_link**	sio	weather
geophone	**power**	**spi**	zlist
gw	pressure	**sr**	
hal	**queue**	**sw**	
humidity	rain	testing	

Fig. 38.15 Result of system composition using only the requirements of the main gateway component as specification. Just 36 out of all 49 modules were included by the engine in the final project (emphasized), correctly leaving out, e.g., drivers for optional sensors, test suites, and interfaces

38.5.2 WSN Sensor Node for Air Quality Monitoring

Also for this application, we followed the flow outlined in Sect. 38.5. The existing application software of the node was developed in C for a real-time embedded OS, ChibiOS. This OS has some important features that help increasing the reliability of the applications. For instance, the APIs of the OS are designed to require minimal parameters and to do just one function, with no options and no error conditions.

Since the application was using an Real-Time Operating System (RTOS), we converted several OS modules into library components so that the system composition engine can consider them when searching for a solution to problem specifications. Besides the RTOS modules, we created library components for several application-specific elements, such as the transducer drivers and some special interfaces required by the OS:

`comm` Application layer of the node communication protocol.
`globals` Global definitions and initialization.
`hwcfg/board` Board-specific configurations, e.g., General-Purpose Input/Output-pin (GPIO) and clock setup, and peripherals check.
`crc8` Helper functions for node interface, e.g., CRC calculation.
`if` Interface functions for the node.
`transport` Transport layer for node interface.
`DHT11` Driver for the temperature and humidity sensor.
`GroveDust` Driver for the particulate matter sensor.
`GroveMQ5` Driver for the gas sensor (H2, LPG, CH4, CO, alcohol).
`GroveMQ9` Driver for the gas sensor (CO, coal gas, LPG).
`sensors` Higher-level abstraction of sensor drivers.
`thRdProbes` Periodic reader for sensor data.

Along with these components, we have created library components for an extensive set of OS modules, e.g.:

`can_lld` STM32 Controller Area Network (CAN) subsystem low-level driver source.
`ext_lld` STM32 EXT subsystem low-level driver source.
`adc_lld` STM32F4xx/STM32F2xx ADC subsystem low-level driver source.
`ext_lld_isr` STM32F4xx/STM32F2xx EXT subsystem low-level driver Interrupt Service Routine (ISR) code.
`hal_lld` STM32F4xx/STM32F2xx Hardware Abstraction Layer (HAL) subsystem low-level driver source.
`stm32_dma` Enhanced Direct Memory Access (DMA) helper driver code.
`pal_lld` STM32L1xx/STM32F2xx/STM32F4xx GPIO low-level driver code.
`i2c_lld` STM32 Inter-Integrated Circuit (I2C) subsystem low-level driver source.
`mac_lld` STM32 low-level Media Access Control (MAC) driver code.
`usb_lld` STM32 Universal Serial Bus (USB) subsystem low-level driver source.
`rtc_lld` Real-Time Clock (RTC) low-level driver.
`sdc_lld` STM32 Secure Digital Card (SDC) subsystem low-level driver source.
`spi_lld` STM32 SPI subsystem low-level driver source.
`gpt_lld` STM32 General-Purpose Timer (GPT) subsystem low-level driver source.
`icu_lld` STM32 Input Capture Unit (ICU) subsystem low-level driver header.
`pwm_lld` STM32 Pulse-Width Modulation (PWM) subsystem low-level driver header.
`serial_lld` STM32 low-level serial driver code.
`uart_lld` STM32 low-level Universal Asynchronous Receiver/Transmitter (UART) driver code.
`adc` ADC Driver code.

`can` CAN Driver code.
`ext` EXT Driver code.
`gpt` GPT Driver code.
`hal` HAL subsystem code.
`i2c` I2C Driver code.
`icu` ICU Driver code.
`mac` MAC Driver code.
`mmcsd` Multimedia/Secure Digital Card (MMC/SD) cards common code.
`mmc_spi` MMC/SD over SPI driver code.
`pal` Input/Output (I/O) Ports Abstraction Layer code.
`pwm` PWM Driver code.
`rtc` RTC Driver code.
`sdc` SDC Driver code.
`serial` Serial Driver code.
`serial_usb` Serial over USB Driver code.
`spi` SPI Driver code.
`tm` Time Measurement driver code.
`uart` UART Driver code.
`usb` USB Driver code.
`chcond` Condition Variables code.
`chdebug` ChibiOS/RT Debug code.
`chdynamic` Dynamic threads code.
`chevents` Events code.
`chheap` Heaps code.
`chlists` Thread queues/lists code.
`chmboxes` Mailboxes code.
`chmemcore` Core memory manager code.
`chmempools` Memory Pools code.
`chmsg` Messages code.
`chmtx` Mutexes code.
`chqueues` I/O Queues code.
`chregistry` Threads registry code.
`chschd` Scheduler code.
`chsem` Semaphores code.
`chsys` System-related code.
`chthreads` Threads code.
`chvt` Time and Virtual Timers related code.
`nvic` Cortex-Mx Nested Vectored Interrupt Controller (NVIC) support code.
`chcore` ARM Cortex-Mx port code.
`chcore_v7m` ARMv7-M architecture port code.
`crt0` Generic ARMvx-M (Cortex-M0/M1/M3/M4) startup file for ChibiOS/RT.
`vectors` Interrupt vectors for the STM32F4xx family.
`chprintf` Mini printf-like functionality.

Figure 38.16 shows an example of a library component that was created for this project. Its structure is similar to the one shown in Fig. 38.14 in terms of dependency and interface data declarations. Besides these, the component includes a parameter definition under the *rpcs* tag. These parameters are made available by the functional code of the component to allow their remote control by a middleware layer or by a monitoring server, using specific protocols. The data associated to these parameters in the library component, which is enclosed in an *rpc* tag, is extracted by the composition tool and is attached to each solution that was generated. These data are later used by external tools for their run-time configuration to properly interface

```
<sgraph:Gss xmi:id="_c3cd36f777bdea1a5ae079">
    <properties>
        <name>comm_component</name>
        <cmt><rpcs><rpc>
        <description>Set sensor sampling frequency.</description>
        <name value="RATE" /><values><set type="integer" /></values>
        </rpc></rpcs></cmt>
    </properties>
    <views xmi:id="_190b8090159699e0b68bce">
        <behavior>
            <view xmi:id="_b860c0e4a20a75489a5f76" name="comm.c"
            type="source" language="C" encoding="base64">
                <mem>LyoNCiAqIENvbW [...]</mem></view>
            <view xmi:id="_37a76b4ed27649239a0554" name="comm.h"
            type="source" language="C" encoding="base64">
                <mem>LyoNCiAqIENvbW0uaA0KICo [...]</mem></view>
        </behavior>
    </views>
    <interfaces>
        <behavior>
            <provide><data>
                <name>m_sPacket</name><base>t_PktHeader</base><size>6</size>
            </data></provide>
            <provide><function>
                <name>Comm_Init</name><return><type>void</type></return>
                <port><ord>1</ord><type>void</type></port>
            </function></provide>
            <provide><function>
                <name>Comm_Write</name><return><type>void</type></return>
                <port><ord>1</ord><type>t_u8 *</type></port>
                <port><ord>2</ord><type>t_u8</type></port>
            </function></provide>
            <require><function>
                <name>Crc8</name><return><type>unsigned char</type></return>
                <port><ord>1</ord><type>void *</type></port>
                <port><ord>2</ord><type>int</type></port>
                <port><ord>3</ord><type>unsigned char</type></port>
            </function></require>
            <require><data>
                <name>g_Kau8Sync</name><base>t_u8</base><size>4</size>
            </data></require>
            <require><data>
                <name>g_pSApp</name><base>SerialDriver *</base><size>4</size>
            </data></require>
        </behavior>
    </interfaces>
</sgraph:Gss>
```

Fig. 38.16 Example of a library component used for the air quality monitoring application. It includes a parameter that can be remotely accessed at run time

with the component. The data can also include human readable descriptions for the developers or the beneficiaries of the WSN application.

These library components were added to the same library that was used for the first application. In this way, the synthesis engine is able to compose systems for both hardware node types by selecting suitable compatible components to match the specification requirements.

For this application, the solver found a suitable system composition with a maximum recursion depth of 109, matching 22 abstract requirements, 50 functional

requirements, and 12 data requirements, in less than 0.2 s on an 1.8 GHz Intel® Core™ i7-2677M processor.

We used the toolset to compose nodes with different sensors, sensor combinations, sensing periods, and remote monitoring interfaces (as the one mentioned above). The synthesis engine used the high-level requirements in the top-level component (which were provided by the developer) to automatically select and compose suitable hardware, software, and configuration for the node.

38.6 Conclusion

Wireless sensor networks can be used for many applications in a variety of domains, but their reliability, lifetime, overall cost, and design effort limit their actual use. Moreover, WSN design flows often lack a well-defined separation between the application designers and the multidisciplinary engineering knowledge needed to cover the operation of the underlying technology. This considerably reduces the use of WSN solutions by the application domain experts, even though WSNs would provide very effective solutions for their applications.

We briefly overviewed some of the most important existing WSN development techniques, abstractions, and tool categories to evaluate how well they respond to these requirements. From the review, the importance of the trade-off between implementation optimization and accessibility to application domain experts became apparent. On the one hand, the development flows that allow significant design optimizations imply a level of hardware, software, and network design knowledge that is seldom found among application domain experts. On the other hand, highly abstracted design flows may often lead to poorly optimized WSN designs and are difficult to port to target platforms outside the (often) narrow range supported by the tool. Also, most of the tools themselves generally lack composability and the ability to be used as building blocks within new development flows.

Model-based design flows seem to provide effective trade-offs between the manual effort that is required to optimize the designs and the availability of a high-level development flow that increases designer productivity. In this context, we presented in more detail two innovative toolsets that offer user-friendly high-level design entry interfaces as well as various degrees of automation to hide the low-level implementation details from the developer. Both flows allow design optimization to various degrees and also manual optimization for skilled developers to increase the performance of the resulting WSN designs. To evaluate their effectiveness, we have illustrated the use of both tools for the development of some typical applications.

References

1. Abrach H, Bhatti S, Carlson J, Dai H, Rose J, Sheth A, Shucker B, Deng J, Han R (2003) MANTIS: system support for multimodAl NeTworks of In-situ Sensors. In: Proceedings of the 2nd ACM international conference on wireless sensor networks and applications, WSNA '03. ACM, New York, pp 50–59. doi:10.1145/941350.941358

2. Antonopoulos C, Asimogloy K, Chiti S, D'Onofrio L, Gianfranceschi S, He D, Iodice A, Koubias S, Koulamas C, Lavagno L, Lazarescu MT, Mujica G, Papadopoulos G, Portilla J, Redondo L, Riccio D, Riesgo T, Rodriguez D, Ruello G, Samoladas V, Stoyanova T, Touliatos G, Valvo A, Vlahoy G (2016) Integrated toolset for WSN application planning, development, commissioning and maintenance: the WSN-DPCM ARTEMIS-JU project. Sensors 16(6):804. doi:10.3390/s16060804
3. Ashton K (2009) That 'Internet of Things' thing. Expert view RFID J http://www.rfidjournal.com/article/view/4986
4. Cao Q, Abdelzaher T, Stankovic J, He T (2008) The LiteOS operating system: towards Unix-like abstractions for wireless sensor networks. In: Proceedings of the 7th international conference on information processing in sensor networks, IPSN '08. IEEE Computer Society, Washington, DC, pp 233–244. doi:10.1109/IPSN.2008.54
5. Cha H, Choi S, Jung I, Kim H, Shin H, Yoo J, Yoon C (2007) RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In: Proceedings of the 6th international conference on information processing in sensor networks, IPSN '07. ACM, New York, pp 148–157. doi:10.1145/1236360.1236381
6. Compton M, Henson C, Lefort L, Neuhaus H, Sheth A (2009) A survey of the semantic specification of sensors. In: 2nd international semantic sensor networks workshop
7. Costa P, Mottola L, Murphy AL, Picco GP (2007) Programming wireless sensor networks with the TeenyLime middleware. In: Proceedings of the ACM/IFIP/USENIX 2007 international conference on middleware, middleware '07. Springer, New York, pp 429–449.
8. Doddapaneni K, Ever E, Gemikonakli O, Malavolta I, Mostarda L, Muccini H (2012) A model-driven engineering framework for architecting and analysing wireless sensor networks. In: Proceedings of the third international workshop on software engineering for sensor network applications, SESENA '12. IEEE Press, Piscataway, pp 1–7
9. Dong W, Chen C, Liu X, Bu J (2010) Providing OS support for wireless sensor networks: challenges and approaches. Commun Surv Tuts 12(4):519–530. doi:10.1109/SURV.2010.032610.00045
10. Dunkels A, Gronvall B, Voigt T (2004) Contiki – a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the 29th annual IEEE international conference on local computer networks, LCN '04. IEEE Computer Society, Washington, DC, pp 455–462. doi:10.1109/LCN.2004.38
11. Eswaran A, Rowe A, Rajkumar R (2005) Nano-RK: an energy-aware resource-centric RTOS for sensor networks. In: Proceedings of the 26th IEEE international real-time systems symposium, RTSS '05. IEEE Computer Society, Washington, DC, pp 256–265. doi:10.1109/RTSS.2005.30
12. Gámez N, Cubo J, Fuentes L, Pimentel E (2012) Configuring a context-aware middleware for wireless sensor networks. Sensors 12(7):8544–8570
13. Gay D, Levis P, von Behren R, Welsh M, Brewer E, Culler D (2003) The nesC language: a holistic approach to networked embedded systems. SIGPLAN Not 38(5):1–11. doi:10.1145/780822.781133
14. Greenstein B, Kohler E, Estrin D (2004) A sensor network application construction kit (SNACK). In: Proceedings of the 2nd international conference on embedded networked sensor systems, SenSys '04. ACM, New York, pp 69–80. doi:10.1145/1031495.1031505
15. Gummadi R, Gnawali O, Govindan R (2005) Macro-programming wireless sensor networks using Kairos. In: Proceedings of the first IEEE international conference on distributed computing in sensor systems, DCOSS'05. Springer, Berlin/Heidelberg, pp 126–140. doi:10.1007/11502593_12
16. Han CC, Kumar R, Shea R, Kohler E, Srivastava M (2005) A dynamic operating system for sensor nodes. In: Proceedings of the 3rd international conference on mobile systems, applications, and services, MobiSys '05. ACM, New York, pp 163–176. doi:10.1145/1067170.1067188
17. Hill J, Szewczyk R, Woo A, Hollar S, Culler D, Pister K (2000) System architecture directions for networked sensors. SIGARCH Comput Archit News 28(5):93–104. doi:10.1145/378995.379006

18. Lazarescu MT (2013) Design of a WSN platform for long-term environmental monitoring for IoT applications. IEEE J Emerg Sel Top Circuits Syst 3(1):45–54. doi:10.1109/JETCAS.2013.2243032
19. Madden SR, Franklin MJ, Hellerstein JM, Hong W (2005) TinyDB: an acquisitional query processing system for sensor networks. ACM Trans Database Syst 30(1):122–173. doi:10.1145/1061318.1061322
20. Mathworks (2013) Generate C and C++ code from simulink and stateflow models. The MathWorks. https://it.mathworks.com/products/simulink-coder/
21. MATLAB and Simulink Release 2010a (2010) The MathWorks, Inc., Natick, Massachusetts, United States
22. MATLAB and Stateflow Release 2010a (2010) The MathWorks, Inc., Natick, Massachusetts, United States
23. Mohamed N, Al-Jaroodi J (2011) A survey on service-oriented middleware for wireless sensor networks. Serv Oriented Comput Appl 5(2):71–85. doi:10.1007/s11761-011-0083-x
24. Mottola L, Picco GP (2011) Programming wireless sensor networks: fundamental concepts and state of the art. ACM Comput Surv 43(3):19:1–19:51. doi:10.1145/1922649.1922656
25. Mottola L, Picco GP (2012) Middleware for wireless sensor networks: an outlook. J Internet Serv Appl 3(1):31–39. doi:10.1007/s13174-011-0046-7
26. Mülder A, Nyßen A (2011) TMF meets GMF. Eclipse Mag 3:74–78. https://svn.codespot.com/a/eclipselabs.org/yakindu/media/slides/TMF_meets_GMF_FINAL.pdf
27. OMG, XML (2007) Metadata Interchange (XMI) Specification. http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm. (Accessed 4 June 2016)
28. Palermo G, Silvano C, Valsecchi S, Zaccaria V (2003) A system-level methodology for fast multi-objective design space exploration. In: Proceedings of the 13th ACM great lakes symposium on VLSI, GLSVLSI '03. ACM, New York, pp 92–95. doi:10.1145/764808.764833
29. Paulon A, Fröhlich A, Becker L, Basso F (2013) Model-driven development of WSN applications. In: 2013 III Brazilian symposium on computing systems engineering (SBESC), pp 161–166. doi:10.1109/SBESC.2013.27
30. Ray A (2009) Planning and analysis tool for large scale deployment of wireless sensor network. Int J Next-Gener Netw (IJNGN) 1(1):29–36
31. Romer K, Mattern F (2004) The design space of wireless sensor networks. IEEE Wirel Commun 11(6):54–61. doi:10.1109/MWC.2004.1368897
32. Shimizu R, Tei K, Fukazawa Y, Honiden S (2011) Model driven development for rapid prototyping and optimization of wireless sensor network applications. In: Proceedings of the 2nd workshop on software engineering for sensor network applications, SESENA '11. ACM, New York, pp 31–36. doi:10.1145/1988051.1988058
33. Sirio G (2013) ChibiOS/RT. http://www.chibios.org/ (Accessed 4 June 2016)
34. Sugihara R, Gupta RK (2008) Programming models for sensor networks: a survey. ACM Trans Sen Netw 4(2):8:1–8:29. doi:10.1145/1340771.1340774
35. Taherkordi A, Loiret F, Abdolrazaghi A, Rouvoy R, Le-Trung Q, Eliassen F (2010) Programming sensor networks using REMORA component model. In: Proceedings of the 6th IEEE international conference on distributed computing in sensor systems, DCOSS'10. Springer, Berlin/Heidelberg, pp 45–62. doi:10.1007/978-3-642-13651-1_4
36. Varga A, Hornig R (2008) An overview of the OMNeT++ simulation environment. In: Proceedings of the 1st international conference on simulation tools and techniques for communications, networks and systems & workshops, Simutools '08. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, pp 60:1–60:10

Codesign Case Study on Transport-Triggered Architectures

39

Jarmo Takala, Pekka Jääskeläinen, and Teemu Pitkänen

Abstract

Application-specific processors are used to obtain the efficiency of fixed-function application-specific integrated circuits and flexibility of software implementations on programmable processors. The efficiency is achieved by tailoring the processor architecture according to the requirements of the application while the flexibility is provided by the programmability. In this chapter, we introduce a hardware/software codesign environment for developing application-specific processors, which is using processor templates based on the transport-triggering paradigm, hence the name transport-triggered architecture (TTA). Fast Fourier transform (FFT) is used as an example application to illustrate the customization. Specific features of FFTs are discussed, and we show how those can be exploited in FFT implementations. We have customized a TTA processor for FFT, and its energy efficiency is compared against several other FFT implementations to prove the potential of the concept.

Acronyms

ADF	Architecture Description File
ASIC	Application-Specific Integrated Circuit
ASP	Application-Specific Processor
CORDIC	COordinate Rotational DIgital Computer
DFT	Discrete Fourier Transfrom
DIF	Decimation-in-Frequency
DIT	Decimation-in-Time
DSP	Digital Signal Processor

J. Takala (✉) • P. Jääskeläinen
Tampere University of Technology, Tampere, Finland
e-mail: jarmo.takala@tut.fi; pekka.jaaskelainen@tut.fi

Teemu Pitkänen
Ajat Oy, Espoo, Finland
e-mail: teemu.pitkanen@ajat.fi

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_39

FFT	Fast Fourier Transform
HDB	Hardware Database
IR	Intermediate Representation
OSAL	Operation Set Abstraction Layer
RTL	Register Transfer Level
TCE	TTA-based Codesign Environment
TTA	Transport-Triggered Architecture

Contents

39.1 Introduction

Application-Specific Processors (ASPs) are used to obtain the efficiency of fixed-function Application-Specific Integrated Circuits (ASICs) and flexibility of software implementations on programmable processors. The efficiency is obtained by tailoring the processor architecture according to the requirements of the application as discussed earlier in ▸ Chap. 12, "Application-Specific Processors" and ▸ Chap. 33, "Hardware/Software Codesign Across Many Cadence Technologies". The simplest customization is to start with existing architecture and remove all the resources, which are not needed to execute the given application. In a similar fashion, an ASP may be constructed from library components, i.e., components are reused; thus the design process is shorter than in ASIC design. Even better efficiency can be obtained if the specific computation patterns in the application are identified and those are converted as accelerators or user-specific function units in the architecture.

The design space for application-specific processors is huge, and finding a suitable architecture for a given application will be an exhaustive work. The optimization strategies for design space exploration are covered in ▸ Chap. 6, "Optimization Strategies in Design Space Exploration" and architecture design space exploration

in ▸ Chap. 8, "Architecture and Cross-Layer Design Space Exploration". Determining the machine code for an arbitrary processor architecture is extremely a difficult and time-consuming task especially when the processor contains parallelism. This work can be alleviated by limiting the search space. Such a constraint can be created by defining a processor template, which provides a set of customization parameters to vary the processor candidates. The limited set of parameters allows retargeting the software development toolchain; thus machine code of the given application can be generated on the customized architecture. The processor customization is an iterative process, thus during each iteration there is a need to port the application program to the new processor. This calls either for a manual assembly language program rewrite or a retargetable compiler, which can adapt to the changes in the architecture.

In this chapter, we introduce a processor template based on transport triggering paradigm and a software/hardware codesign environment supporting this template. We illustrate the use of the template and design environment by using Fast Fourier Transform (FFT) as an example application. FFT is used to compute Discrete Fourier Transfrom (DFT) of a sequence, which in turn converts the time domain representation of a digital signal to a frequency domain representation. The definition of DFT contains redundancy, and several methods have been proposed to avoid these redundancies. In general any method for computing DFT with lower arithmetic complexity than DFT is called a fast Fourier transform. FFT has been considered as "the most important numerical algorithm of our lifetime" [36], and nowadays it has gained popularity as frequency division has been used in many modern wireless communications standards.

In this chapter, we discuss FFT algorithms and illustrate some of their properties, which can be exploited when implementing the transform. This chapter shows how these properties can be exploited in implementations and a processor tailored for FFT is described. The energy efficiency of the tailored processor is compared against several implementations from the literature to show the efficiency of the approach.

39.2 Transport-Triggered Architecture Template

In this chapter, we use exposed data path as one of the characteristics of the architectural template, i.e., a template where many processor data-path details are visible to the programmer who can directly control those resources. Examples of such architectures are, e.g., MOVE [11], MOVE-Pro [18], FlexCore [42], STA [8], and ELM [12]. In particular, we exploit transport triggering paradigm [10], which defines that operation execution is initiated by data transport rather than operation defining the data transports as in traditional programming models. In Transport-Triggered Architecture (TTA) programming model, the program defines only data moves and the operations occur as side effects of data transports. In a way, transport triggering evokes the traditional data-flow model of execution. Operands to a function unit are moved via an interconnection network to input ports, and one of

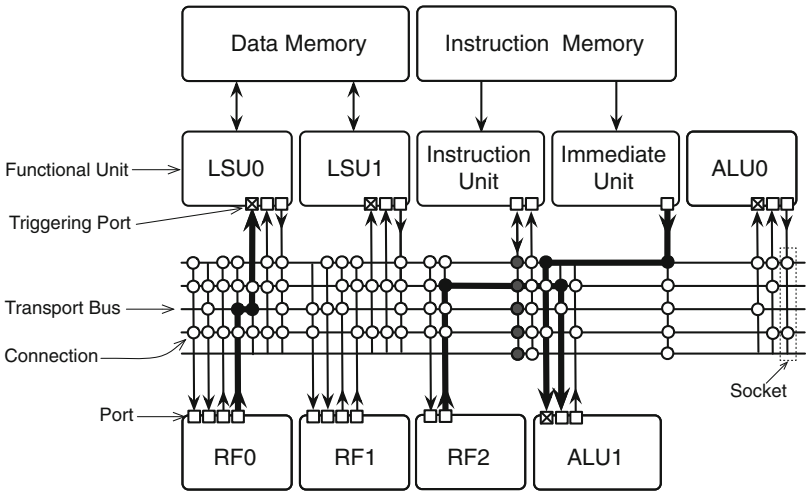


Fig. 39.1 TTA processor organization

the ports is dedicated as a trigger. Whenever data is moved to the trigger port, the operation execution is initiated. The program defines only the data moves on the interconnection network; thus the TTA processor has only one instruction: move. As the program defines moves in the interconnection network, the TTA processors have a programmer-exposed interconnection network.

An example TTA processor is depicted in Fig. 39.1. The interconnection network in this processor contains five transport buses implying that at most five data transports can be executed simultaneously. This also implies that each instruction contains five move slots, where each slot specifies the data transport carried out in each bus. The figure illustrates execution of an instruction with three parallel moves, i.e., instruction has three move slots:

$$\#4 \rightarrow \text{ALU1.i0.ADD; RF2.r3} \rightarrow \text{ALU1.i1; RF0.r1} \rightarrow \text{LSU0.i0.STW}$$

On the first transport bus, an immediate value is moved to the input port 0 of the function unit ALU1. The immediate value is actually obtained from the immediate unit, which has only one output port. As the function units can perform several operations, the move carries also information about the operation to be executed; opcode ADD is transported to function unit along with the operand. The second bus transports an operand from register r3 through the output port 0 of the register file RF2 to the input port 1 of the ALU1. The third bus is used to transport a value from register r1 in the register file RF0 through the output port 1 to the input port 0 of the load-store unit LSU0. The third move contains an opcode indicating that the transported word is to be stored to memory. The actual store address has been defined by another move to port 1 of the LSU0. The remaining two move slots are empty; thus the corresponding two buses are not used in this instruction. Thus they can be considered executing a NOP.

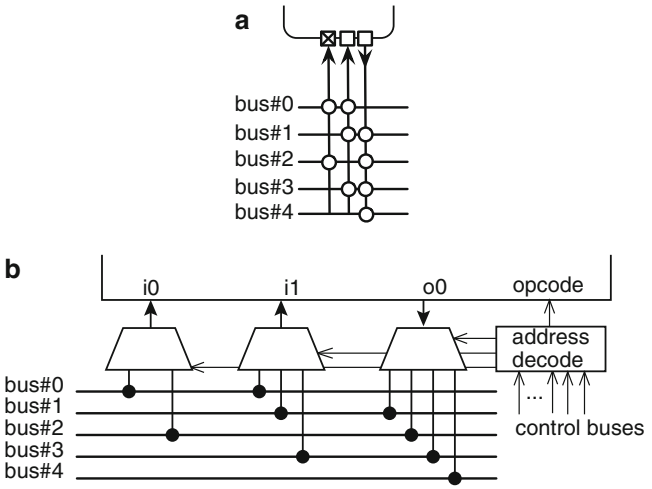


Fig. 39.2 Principal socket interface for function units: (**a**) high-abstraction-level representation and (**b**) structure

Figure 39.1 shows that the instructions control the operation of each transport bus through the instruction unit. The connection to each bus convoys control information, e.g., the source and destination of the transport move, possible opcode for the operation to be executed, etc. The function units are connected to the transport buses with the aid of sockets. The interconnection network in the architectural template consists of buses and sockets. The principal concept of sockets is illustrated in Fig. 39.2; each port of a function unit has a socket, which defines the connections to the buses. When the control information in a bus indicates that the port is the destination for the current move instruction, data from the bus is passed to the port. In a similar fashion, data from the source port is forwarded to the bus.

The architecture template defines that one of the input ports is a trigger port and a move to this port triggers the specified operation. The concept is illustrated in Fig. 39.3, where the trigger port is indicated by a cross in the input port. It should be noted that a function unit has only one trigger port. A move to this port will latch data from the bus to trigger register, and the operation execution starts with operands from the trigger port and other operand registers; the function unit in Fig. 39.3 expects two operands; thus there is one trigger register and one operand register. The operand to the operand register can be moved by an earlier instruction. The operand can also be moved in the same instruction as the trigger port moves if there are buses available to carry out the move. In Fig. 39.1, the first bus is used to transport an operand to trigger port of ALU1. The second bus moves the other operand from RF3; thus the operands for the specified ADD operation are moved to function unit at the same cycle. The move over the third bus triggers the store operation, but the actual store address has been moved to the input port 1 of LSU0 by an earlier instruction.

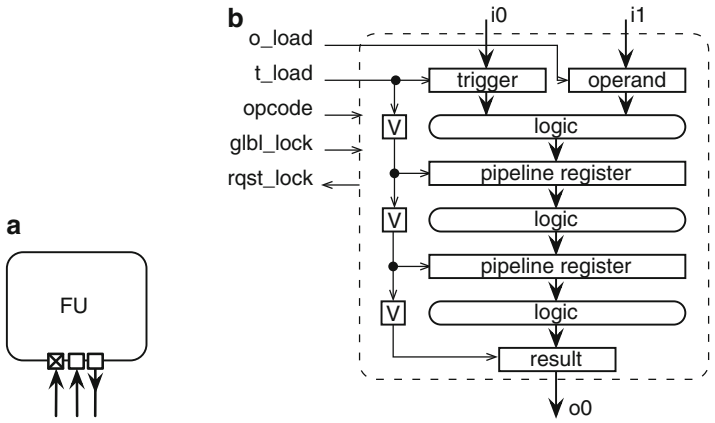


Fig. 39.3 Pipelined function unit based on semi-virtual latching: (**a**) high-abstraction-level representation and (**b**) principal block diagram. The result register in the output is optional

In TTAs, function units can be pipelined, and in the template, semi-virtual time latching [10] method is used, where valid bits control the pipeline as depicted in Fig. 39.3b. The pipeline starts an operation whenever there is a move to the trigger port, i.e., the *o_load* signal is active. As valid bits control the pipeline, a single pipeline stage is active only once for one trigger move. For example, Fig. 39.3b illustrates a case, where the result can be read from the result register three instructions after the trigger move.

An external or internal event can lock the processor (*glbl_lock* signal is active); all the function units in the processor have their pipelines stalled. The architectural template requires each operation in a function unit to have a deterministic latency such that the result read for the operation can be scheduled properly. If the function unit faces an unexpected longer latency operation, e.g., a memory refresh cycle or a function unit has iterative operation of which latency depends on the inputs, the unit can request the processor to be locked by activating the *rqst_lock* signal until the ongoing operation is completed.

In traditional statically scheduled machines, the timing between operand load, operation execution, and result store is fixed at design time. In TTAs, the timing is defined at compile time. For example, multiplication instruction defines completely when the operands are read from the registers R0 and R1 and result is stored to register R3:

MUL R3, R2, R1

while in TTA the corresponding operation can be specified with three different moves. When assuming a single transport bus and a function unit, which performs only multiplication and that the input port 0 is the trigger port, the previous instruction would be:

RF.r2 → MUL.i1;
RF.r1 → MUL.i0;
MUL.o0 → RF.r3

or if two transport buses are available:

RF.r2 → MUL.i1; RF.r1 → MUL.i0;
MUL.o0 → RF.r3;

However, the moves can even be scheduled over a large block of instructions:

RF.r2 → MUL.i1; . . . ; . . . ;
⋮
. . . ; . . . ; RF.r1 → MUL.i0;
⋮
MUL.o0 → RF.r3; . . . ; . . . ;

The previous examples show that there is a high degree of freedom on scheduling the moves over move slots in neighboring instructions compared to traditional statically scheduled machines, which makes the TTA scheduling a challenging problem.

The TTA instruction format reminds horizontal microcode, which usually shows poor instruction density. However, experiments show that the instruction overhead due to the exposed data-path control is negligible when comparing to the savings if the workload is data-intensive and the interconnection network is carefully optimized [21,43].

The exposed data-path template opens unique optimization opportunities. For example, due to the explicit result transfers, the function units are independently executing isolated modular components in the data path. In the point of view of processor design methodology, the modularity allows point-and-click style tailoring of the data-path resources from existing processor component databases. It also means the function units can have arbitrary latencies and pipeline lengths from a single cycle because there is no hazard detection hardware. There is no practical limit to the number of outputs produced by operations.

The TTA template allows the processor to be customized in various ways. User can define the sets of basic TTA components to be included in the architecture. The number of register files can be varied, and each register file can have additional specifications: the number of registers, word width, and the number of read/write ports. Function units can be tailored by varying the number of function units, and for each function unit, it is possible to define the operation set implemented by the function unit, the number of input and output ports, the width of the ports, resource sharing/pipelining, and accessed address space (in case of a load-store unit). The operation set can be varied, and for each operation the number of operands, the number and data type of results and operands, and operation state data can be parametrized. Instruction encoding can be varied and the core can support a number of instruction formats. For each instruction format, the immediate (constant) support can be determined. Parameters related to address spaces include the number of address spaces. For each address space, size, address range, and the numerical id (referred to from program code) can be varied. Finally the parametrization allows even specification of multi-core systems.

An interesting customizable aspect in TTA processors is the interconnection network. As it is visible to the programmer, user can carefully tailor the connectivity according to the application. Another useful feature is the support for multiple disjoint address spaces: one can add one or more private address spaces for local memories inside a core that can be accessed using address space type qualifier attributes in the input C code.

39.3 Design Flow for Customizing Transport-Triggered Architectures

The design work described in this chapter is carried out with the TTA-based Codesign Environment TTA-based Codesign Environment (TCE) [40]. The codesign process supported by the TCE tools is illustrated in Fig. 39.4. Initially, the designer has a set of requirements and goals placed to the end result.

The iterative customization process starts with an initial predesigned architecture, which contains minimal resources to compile an arbitrary C program to run on the machine. The designer can add, modify, and remove architecture components using a graphical user interface tool called Processor Designer (ProDe) shown in Fig. 39.5, which creates the processor description in Architecture Description File (ADF) format. Each iteration of the processor can be evaluated by compiling the application code on the architecture with retargetable high-level language compiler and simulating the resulting parallel assembly code with the instruction-set simulator.

The simulator shows statistics of the run time of the program and the utilization of the different data-path components, indicating bottlenecks in the design. The processor simulator provides a compiled simulation engine for fast evaluation cycles and a more accurate interpretive engine for software debugging which supports common software debugging features such as breakpoints.

An essential feature in the processor customization process is the inclusion of custom operations. For a completely new processor operation, the designer describes the operation simulation behavior in C/C++ to Operation Set Abstraction

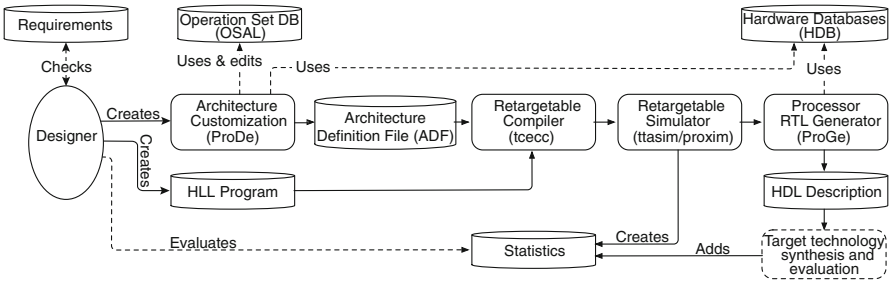


Fig. 39.4 TCE design flow for tailoring TTA processors

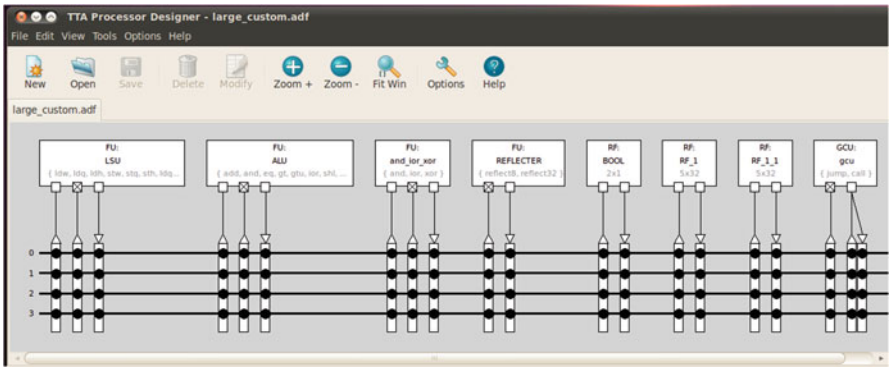


Fig. 39.5 Graphical user interface for ProDe tool

Layer (OSAL) database, estimates its latency in instruction cycles when implemented in hardware, and adds the operation to one of the function units in the architecture. This way it is possible to see the effects of the custom hardware to the cycle count, before deciding whether to include it in the design or not.

When a design point fulfilling the requirements has been found, or more accurate statistics of a design point is needed, the designer can generate synthesizable Register Transfer Level (RTL) description of the processor with processor generator (ProGe). For this step, the designer has to add RTL descriptions of the user-specific custom function units to Hardware Database (HDB). In order to alleviate this process, the function unit implementation is automatically verified against its architecture simulation model. The generated RTL can be synthesized with third party synthesis and simulation tools to obtain more detailed statistics of the processor. The TCE environment has an automated process to optimize the interconnection network, e.g., merging buses [43], removing function units until the performance does not increase, or removing connections which do not decrease the performance. The connectivity between components in larger TTA designs is hard to manage manually due to the huge space of options.

Manual assembly language coding would be the last optimization step after the final processor architecture has been selected. During the design process, however, assembly language is not feasible due to the architecture iteration process; whenever the architecture is changed, the affected parts of the assembly code would need to be rewritten.

In general, high-level language compilers cannot automatically exploit the complex custom operations in the processor for accelerating the program execution. Often compilers cannot extract all the inherent parallelism from the program description to exploit all the parallel processor resources. As high-level programming is typically preferred, the key tool in TCE is the retargetable software compiler, tcecc. The compiler uses LLVM [29] compiler framework as a backbone. The compiler supports C/C++ languages and has also support for the parallel OpenCL standard [22], in particular with *pocl* library [23]. The frontend supports the ISO

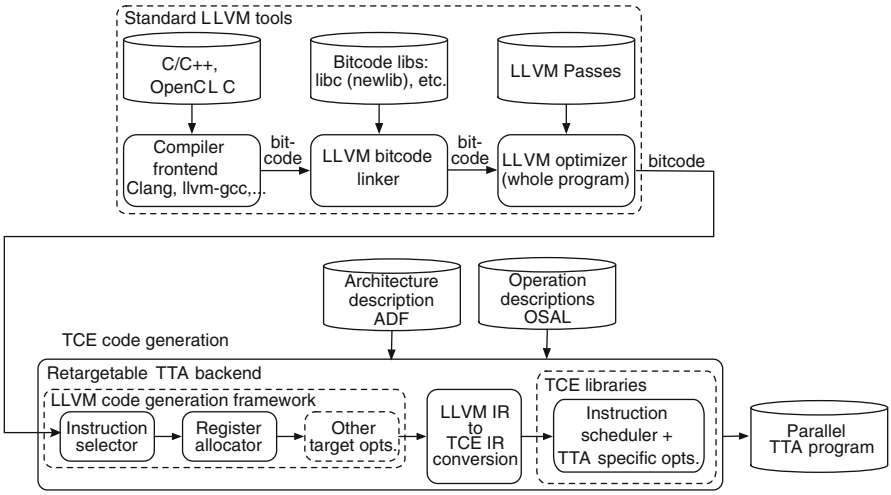


Fig. 39.6 tcecc compiler

C99 standard with a few exceptions, most of the C++98 language constructs, and a subset of OpenCL C. Although TCE tools support multi-threading and multi-core systems [24], in this chapter we limit the discussion to single thread operation.

The main compilation phases of tcecc are shown in Fig. 39.6. Initially, LLVM's Clang frontend converts the source code to the LLVM internal representation. After the frontend has compiled the source code to LLVM bytecode, the utility software libraries are linked in, producing a fully linked self-contained bytecode program. Then standard LLVM Intermediate Representation (IR) optimization passes are applied to the bytecode-level program, and the whole-program optimizations can be applied aggressively. The optimized bytecode is then passed to the TCE retargetable code generation.

User-specific custom operations can be described in OSAL database as data-flow graphs consisting of primitive operations, which the LLVM instruction selector automatically attempts to detect and replace in the program code. Complex custom operations consisting of several primitive operations and dependencies between them or custom operations producing multiple results may not be automatically detected from intermediate code. Therefore, tcecc produces intrinsics that can be used manually in the source code.

39.4 Discrete Fourier Transform and Its Fast Algorithms

DFT is used to convert a finite sequence of equally spaced samples to a sequence of coefficients of a finite combination of complex sinusoids. In other words, the time domain representation of an N-point discrete time signal $x(n)$ is converted to frequency domain representation $X(r)$ as follows [31]:

$$X(r) = \sum_{n=0}^{N-1} x(n) W_N^{rn},\ r = 0, 1, \cdots, N-1, \tag{39.1}$$

where the coefficients W_N are defined as

$$W_N = e^{-j2\pi/N} = \cos(2\pi/N) - j \sin(2\pi/N), \tag{39.2}$$

where j denotes the imaginary unit. As the coefficients W_N are composed of sine and cosine functions, the coefficients W_N^{rn} have symmetry and periodicity properties, which implies that the DFT defined in (39.1) contains redundancy. By exploiting the underlying properties of the coefficients W_N^{rn}, several fast algorithms for DFT, i.e., FFTs, have been developed over the years. The most popular FFT is the Cooley-Tukey algorithm [9], where divide and conquer paradigm is used to decompose DFT into a set of smaller DFTs. In particular, the Cooley-Tukey principle states that a DFT of length $N = PQ$ can be computed with the aid of P-point DFT and Q-point DFT.

39.4.1 Radix-*p* Algorithms

If a factor N is not a prime, the Cooley-Tukey principle can be recursively applied and the larger DFT will be computed with the aid of several smaller DFTs. Especially, when the DFT length is a power of a prime, i.e., $N = p^q$, then the N-point DFT can be computed with the aid of p-point DFTs constructed in q computing stages. As the resulting fast algorithm contains only p-point DFTs, it is called a radix-p FFT. The most popular approach is radix-2 FFT algorithm, where the DFT is decomposed recursively until the entire algorithm is computed with the aid of 2-point DFTs as follows:

$$\begin{aligned} X(r) &= \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_N^{2nr} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_N^{2nr+1} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_{\frac{N}{2}}^{2nr} + W_N^r \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_{\frac{N}{2}}^{2nr},\ r = 0, 1, \cdots, N-1. \end{aligned} \tag{39.3}$$

This equation shows coefficients

$$W_N = e^{-j2\pi/N} \tag{39.4}$$

for the N-th root of unity. Its powers are referred to as *twiddle factors*.

The DFT decomposition can be carried out with two principal approaches: Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF). In DIT approach,

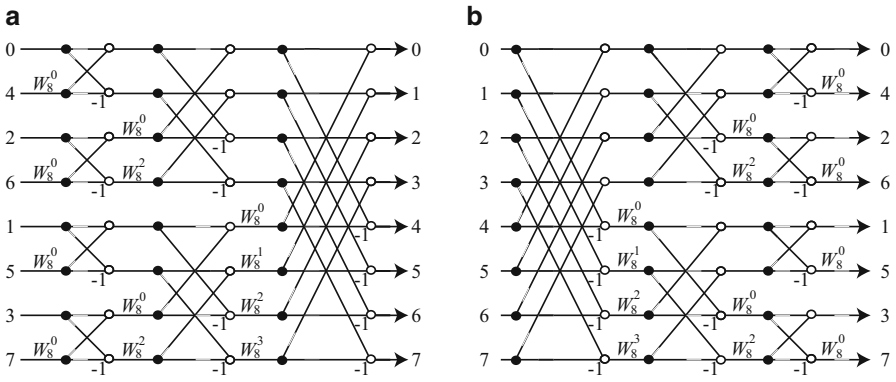


Fig. 39.7 Signal-flow graphs of 8-point radix-2 FFT: (**a**) decimation-in-time and (**b**) decimation-in-frequency algorithm. Circles represent addition

the decomposition is started in time domain sequence, while in DIF approach, the decomposition is started on frequency domain sequence. Both the approaches are illustrated in Fig. 39.7, where the signal-flow graphs of 8-point FFT derived with both approaches are shown. In the 8-point transform, the computations are carried out in three computing stages, where each column contains four 2-point DFTs. The principal computation building block in the radix-2 FFT is 2-point DFT in (39.3), which is also called a radix-2 butterfly. In Fig. 39.7, the weights -1 and W_N^r denote multiplication.

This work concentrates on the DIT approach, while both the DIT and DIF approaches result in the same arithmetic complexity but can be some other implementation-related differences. When implementing the algorithms with fixed-point arithmetic, there will be difference in the numeric accuracy due to quantizations carried out during the computations. Although the differences in signal-to-noise ratio (SNR) can be small, the DIT approach will result in better SNR in radix-2 algorithms [2]. Therefore, in this chapter, we exploit the DIT algorithms.

In the radix-2 butterfly, one complex multiplication and two complex additions are needed, each stage contains $N/2$ butterflies, and the number of the stages is $\log_2 N$, which gives the total of $\frac{N}{2}\log_2 N$ complex multiplications and $N \log_2 N$ additions for an N-point transform.

39.4.2 Radix-2^r Algorithms

Traditionally the most popular FFT have been the radix-2 FFTs, where computations are based on 2-input, 2-output butterflies depicted in Fig. 39.8. The radix-2 FFT is a special case in the class of radix-2^s FFTs [7]. The arithmetic complexity of FFT can be reduced by using greater than two radix if many of the complex coefficients turn out to be trivial (± 1 or $\pm j$). Let us consider the basic equation of the DFT in

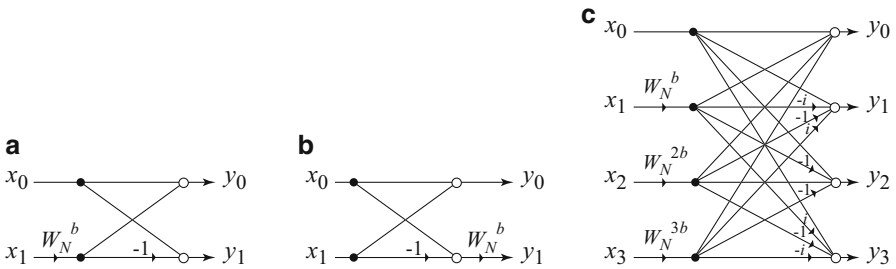


Fig. 39.8 FFT butterflies according to (**a**) radix-2 DIT algorithm in Fig. 39.7a, (**b**) radix-2 DIF algorithm in Fig. 39.7b, and radix-4 DIT algorithm in Fig. 39.9b

(39.1) and divide the original N-point problem to four partial sums by dividing the system to four sub problems, where the length of problem is $N/4$:

$$\begin{aligned} X(r) &= \sum_{n=1}^{N-1} x(n) W_N^{rn} \\ &= \sum_{n=0}^{N/4-1} x(4n) W_N^{r(4n)} + \sum_{n=0}^{N/4-1} x(4n+1) W_N^{r(4n+1)} + \sum_{n=0}^{N/4-1} x(4n+2) W_N^{r(4n+2)} \\ &\quad + \sum_{n=0}^{N/4-1} x(4n+3) W_N^{r(4n+3)}, r = 0, 1, \cdots, N-1. \end{aligned} \tag{39.5}$$

This method results in a radix-4 algorithm, where computations are based on 4-point DFT. This approach has benefits in terms of arithmetic complexity as 4-point DFT can be computed with trivial coefficients. In matrix form, the 4-point and 2-point DFT, F_4 and F_2, respectively, can be defined as

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}; F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{39.6}$$

While the radix-2 FFT has $\log_2 N$ computing stages, the radix-4 algorithm has only $\log_4 N$ stages, which results in significant savings in arithmetic complexity; e.g., a 64-point FFT can be computed in three stages while the radix-2 algorithm requires six computing stages. The arithmetic complexity for an N-point radix-4 FFT is $\frac{3N}{4} \log_4 N$ complex multiplications and $3N \log_4 N$ complex additions. The savings in multiplications (twiddle factors) are illustrated in Fig. 39.9.

From the implementation point of view, the lower number of arithmetic operations provides potential for faster computation and energy savings. In addition,

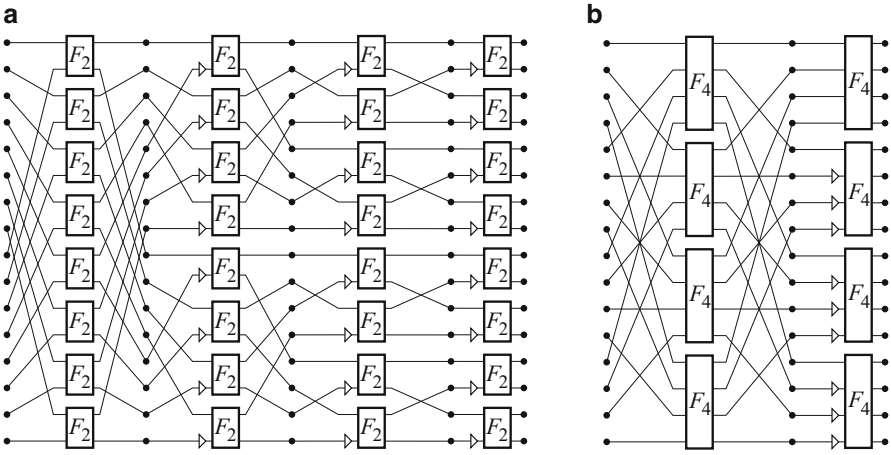


Fig. 39.9 Signal-flow graphs of in-place DIT FFTs: 16-point (**a**) radix-2 and (**b**) radix-4 FFT. Triangles represent a non-trivial twiddle factor

the latency of computations can be shorter. Besides lower arithmetic complexity, the radix-4 FFT provides also other advantages. The lower number of butterfly computation stages implies that, in memory based systems, less memory accesses are required. This speeds up the computations, reduces energy consumption, and relaxes the memory bandwidth requirements.

In this chapter, we exploit the in-order input, permuted output DIT radix-4 FFT defined as

$$F_{2^{2n}} = R_{2^{2n}} \left[\prod_{s=n-1}^{0} [P_{2^{2n}}^{s}]^{T} (I_{2^{(2n-2)}} \otimes F_4) D_{2^{2n}}^{s} P_{2^{2n}}^{s} \right], \tag{39.7}$$

where R_N is a permutation matrix defined as

$$R_{4^n} = \prod_{k=2}^{n} I_{4^{(n-k)}} \otimes P_{4^k,4}, \tag{39.8}$$

P_N^s is a permutation matrix of order N defined as

$$P_{2^n}^{s} = I_{4^s} \otimes P_{2^{(n-2s)},2^{(n-2s-2)}}, \tag{39.9}$$

and D_N^s is a diagonal matrix containing $N = 4^n$ twiddle factors as follows:

$$D_N^s = Q_N^s \left[\bigoplus_{k=0}^{N/4-1} \operatorname{diag}\left(1, W_{4^{s+1}}^{\lfloor k2^{s+1}/N \rfloor}, W_{4^{s+1}}^{2\lfloor k2^{s+1}/N \rfloor}, W_{4^{s+1}}^{3\lfloor k2^{s+1}/N \rfloor}\right) \right]; \tag{39.10}$$

$$Q_N^s = \prod_{k=0}^{s} P_{4^{(s-k)},4} \otimes I_{N/4^{(s-k)}} \, . \tag{39.11}$$

In the previous, $\otimes$ denotes tensor product, i.e.,

$$\begin{pmatrix} 0 & 1 \\ 2 & 1 \end{pmatrix} \otimes A = \begin{pmatrix} 0 & A \\ 2A & A \end{pmatrix} \tag{39.12}$$

and $\oplus$ denotes direct sum, i.e.,

$$A \oplus B = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix}. \tag{39.13}$$

An example of FFT from (39.7) is depicted in Fig. 39.10.

The arithmetic complexity can further be reduced by selecting on even higher radix. However, in radix-8 and higher, the butterflies will contain nontrivial coefficients, and therefore, the relative arithmetic complexity is not decreasing as much. While radix-8 computations are applicable as they provide some advantages in specific implementation styles, higher radices are seldom used.

The main drawback of the radix-p FFTs is the fact that the length of the transform has to be a power of radix, $N = p^s$; i.e., radix-4 algorithms can only be applied when the transform length is a power of four. When the radix is higher, there are less sequence sizes where the algorithm can be applied. Due to this fact, radix-2 FFTs have been popular.

39.4.3 Mixed-Radix FFT

A method to reduce the arithmetic complexity compared to radix-2 FFT but still to support power of two transform lengths is mixed-radix approach, where the DFT decomposition contains several radices, e.g., the results of a 32-point FFT can be computed with two radix-4 stages and a single radix-2 stage. An example of mixed-radix FFT is shown in Fig. 39.11, where the signal-flow graph of a 32-point in-order input, permuted output DIT FFT based on radix-4 and radix-2 is illustrated.

In this chapter, we exploit the mixed-radix approach consisting of radix-4 and radix-2 computations, which provides best of the both worlds: lower arithmetic complexity of radix-4 FFTs and support for all the power-of-two transform sizes of radix-2 FFTs. The mixed-radix FFT consisting of radix-4 processing columns followed by a single radix-2 column can be defined as

$$F_{2^{2n+1}} = O_{2^{(2n+1)}} \left(I_{2^{2n}} \otimes F_2\right) C_{2^{(2n+1)}} \left[\prod_{s=n-1}^{0} [P^s_{2^{(2n+1)}}]^T (I_{2^{(2n-1)}} \otimes F_4) D^s_{2^{(2n+1)}} P^s_{2^{(2n+1)}} \right], \tag{39.14}$$

where the matrices P^s_N and D^s_N are defined in (39.9) and (39.10), respectively. The matrix O_N is a permutation matrix given as

$$O_N = \left(I_2 \otimes R_{4^n}\right) P_{N,2}\ , N = 2^{2n+1}. \tag{39.15}$$

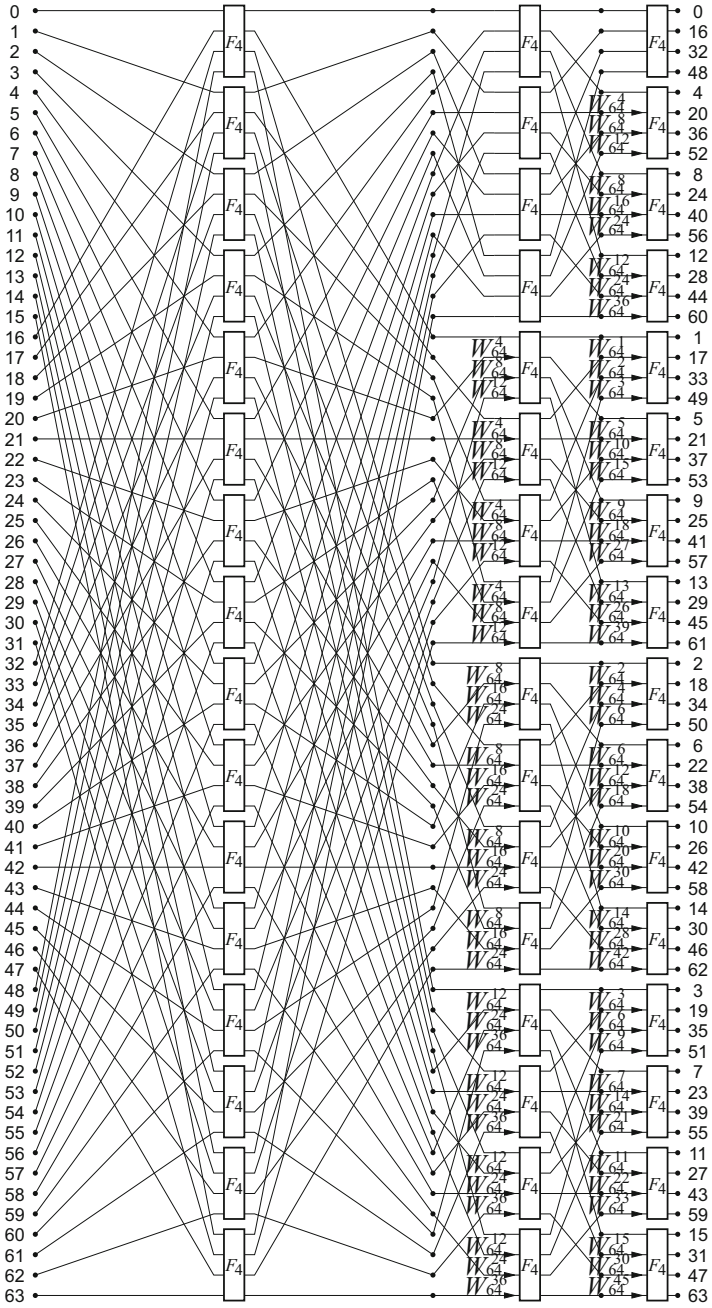


Fig. 39.10 Signal-flow graph of 64-point radix-4 DIT FFT. Numbers in the processing columns denote exponent k of twiddle factor, W_{64}^{k}

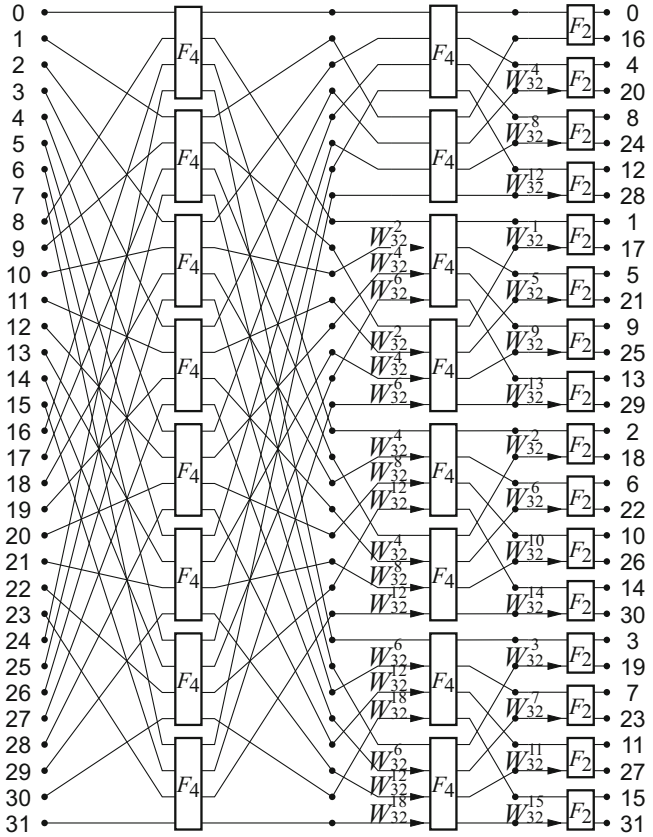


Fig. 39.11 Signal-flow graph of 32-point mixed-radix DIT in-place FFT algorithm

The matrix C_N contains the twiddle factors for the radix-2 processing stage, and it is defined as

$$C_N = Q_N^{\log_4(N/2)} \bigoplus_{k=0}^{N/2-1} \text{diag}\left(1, W_N^k\right), N = 2^{2n+1}, \tag{39.16}$$

where the permutation matrix Q_N^s is defined in (39.11).

The mixed-radix approach allows us to design a system supporting multiple power-of-two FFT sizes. For example, in order to support the IEEE 802.16.1 OFDMA PHY [20], 256-point and 2048-point FFT transforms have to be realized, but the radix-4 FFT cannot be used to compute a 2048-point FFT, while mixed-radix approach is usable.

39.5 Building Blocks and Optimizations

In TTAs, application-specific function units could be exploited. One possible candidate for a special unit can be found by chaining up the operations executed; if an operation pattern is repeated in the application, it is a good candidate for a user-specific function unit. Such operation patterns can be, e.g., memory address generation, complex-valued addition, and complex-valued multiplication. The special units can also contain more specialized and complex functions like twiddle factor generator. The TTA template supports different latencies; thus the special function units can be pipelined to an arbitrary number of stages.

In this section, we discuss several properties of the previous FFT algorithms, which can be exploited when implementing the algorithm. In particular, the special features are used to construct user-specific functional units, which can be used in a TTA processor to speed up FFT computations.

39.5.1 In-Place Computations

In general, FFT algorithms are block processing algorithms, where computing is performed in processing stages consisting of butterfly computations. This is depicted in the signal-flow graphs of the algorithms, e.g., Figs. 39.9 and 39.11. Often in software implementations, double buffering [16] is used, i.e., operands are stored in an array and results are stored to another array and the role of buffers is exchanged for the next iteration. However, the previous signal-flow graphs illustrate that after the input operands for the butterfly operations are available and read from the memory locations, those operands are not needed any more, and the corresponding memory locations can be used to store the results of the butterfly. The results are used as operands for the butterflies in the following computing stage, i.e., computations can be performed in-place [26]. Exploitation of this property reduces significantly the memory requirements of software implementations.

39.5.2 Permutations and Operand Access

The FFT computations can be divided in butterfly computations, e.g., radix-2 FFT shown in Fig. 39.9a consists of 2-point butterfly computations, which each requires two operands and produces two results. The operands for butterflies are obtained with stride access, i.e., if the input sequence is stored in a memory array in order, the operands for butterfly computations in the first computing stage are located $N/2$ apart in the memory. In the second stage, the operands are $N/4$ apart. In software implementations, the operand index computation requires arithmetic operations, but, in application-specific implementations, this can be realized with lower complexity. When investigating the index addressing at bit level, it can be noted that addresses $N/2$ apart can be obtained from a linear address simply with the aid of rotation [7].

A linear address $(a_{N-1}, a_{N-2}, \ldots, a_0)$ is rotated to the right to obtain the operand access index $(a_0, a_{N-1}, a_{N-2}, \ldots, a_1)$. For example, the 2nd butterfly in the first processing stage in Fig. 39.9a reads operands from addresses 1 and 9; thus the mappings are $2_{10} = 0010_2 \rightarrow 1_{10} = 0001_2$ and $3_{10} = 0011_2 \rightarrow 9_{10} = 1001_2$. It should be noted that the access pattern of the operands for butterfly computations depends on the butterfly state s in the FFT signal-flow graph.

Different FFT algorithms have different operand access patterns. The operand indices for the first two processing stages of 64-point radix-4 DIT FFT in Fig 39.10 are listed in Fig. 39.12. The figure shows the decimal and binary representations of the indices, which reveal that the address mapping from linear address to operand index is a rotation. In the first stage shown in Fig. 39.12a, the bit-level mapping is rotation of two bits to the right in a 6-bit address. In the second stage listed in Fig. 39.12b, we can still see the same rotation of two bits to the right, but at this time the field to be rotated contains only four bits. This can be extended to a systematic method illustrated in Fig. 39.13; operand address mapping in 2^{2k}-point radix-4 DIT FFT in bit level is a rotation of two bits to the right in the $(2(k-s))$ least significant bits in the $2k$-bit linear address.

The mixed-radix approach uses yet another mechanism. Let us consider the mixed-radix FFT in Fig. 39.11. It should be noted that the length of the transform is now $N = 2^{2k+1}$, i.e., the index has an odd number of bits. The operand access sequence in the first processing stage is listed in Fig. 39.14a, which shows that the mapping in the bit level is rotation of two bits to the right in the 5-bit address. The addressing sequence in the second processing stage is listed in Fig. 39.14b, which indicates that the mapping is again 2-bit rotation, but the bit field to be rotated contains the three least significant bits in the address. The systematic mapping for mixed-radix FFT defined in (39.14) is illustrated in Fig. 39.15: operand address

a

Linear idx		Operand idx	
0	000000	0	000000
1	000001	16	010000
2	000010	32	100000
3	000011	48	110000
4	000100	1	000001
5	000101	17	010001
6	000110	33	100001
7	000111	49	110001
8	001000	2	000010
9	001001	18	010010
10	001010	34	100010
11	001011	50	110010
12	001100	3	000011
13	001101	19	010011
14	001110	35	100011
15	001111	51	110011

b

Linear idx		Operand idx	
0	000000	0	000000
1	000001	4	000100
2	000010	8	001000
3	000011	12	001100
4	000100	1	000001
5	000101	5	000101
6	000110	9	001001
7	000111	13	001101
8	001000	2	000010
9	001001	6	000110
10	001010	10	001010
11	001011	14	001110
12	001100	3	000011
13	001101	7	000111
14	001110	11	001011
15	001111	15	001111

Fig. 39.12 Operand address sequences 64-point FFT in Fig. 39.10: (**a**) the first computation stage, $s = 0$, and (**b**) the second computation stage, $s = 1$

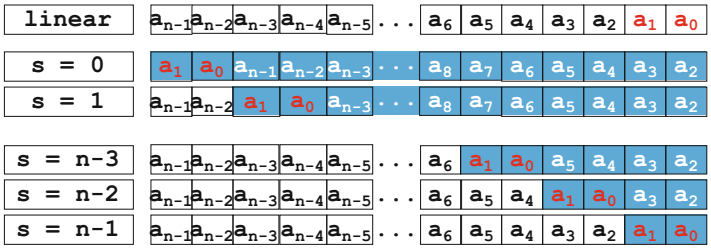


Fig. 39.13 Bit-level operand address mapping for a 2^{2k}-point in-order, permuted output radix-4 DIT FFT. $n = 2k$

a

Linear idx		Operand idx	
0	00000	0	00000
1	00001	8	01000
2	00010	16	10000
3	00011	24	11000
4	00101	1	00001
5	00110	9	01001
6	00110	17	10001
7	00111	25	11001
8	01000	2	00010
9	01001	10	01010
10	01010	18	10010
11	01011	26	11010
12	01100	3	00011
13	01101	11	01011
14	01110	19	10011
15	01111	27	11011

b

Linear idx		Operand idx	
0	00000	0	00000
1	00001	2	00010
2	00010	4	00100
3	00011	6	00110
4	00100	1	00001
5	00101	3	00011
6	00110	5	00101
7	00111	7	00111
8	01000	8	01000
9	01001	10	01010
10	01010	12	01100
11	01011	14	01110
12	01100	9	01001
13	01101	11	01011
14	01110	13	01101
15	01111	15	01111

Fig. 39.14 Operand address sequences for 32-point mixed radix-4 and radix-2 FFT in Fig. 39.11: (**a**) the first computation stage, $s = 0$, and (**b**) the second computation stage, $s = 1$

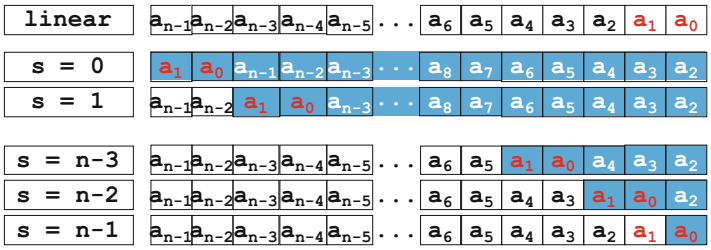


Fig. 39.15 Bit-level operand address mapping for a 2^{2k+1}-point in-order, permuted output mixed radix-4 and radix-2 DIT FFT. $n = 2k + 1$

mapping in 2^{2k+1}-point mixed-radix DIT FFT in bit level is a rotation of two bits to the right in the $(2(k-s)+1)$ least significant bits in the $(2k+1)$-bit linear address.

There is yet another address mapping-related property in FFTs; the transforms contain permutations either in input or output or both. In radix-2 FFTs, the input or output permutations are the well-known bit-reversed permutations as seen in

a

Linear idx		Operand idx	
0	000000	0	000000
1	000001	16	010000
2	000010	32	100000
3	000011	48	110000
4	000100	4	000100
5	000101	20	010100
6	000110	36	100100
7	000111	52	110100
8	001000	8	001000
9	001001	24	011000
10	001010	40	101000
11	001011	56	111000
12	001100	12	001100
13	001101	28	011100
14	001110	44	101100
15	001111	60	111100
16	010000	1	000001

b

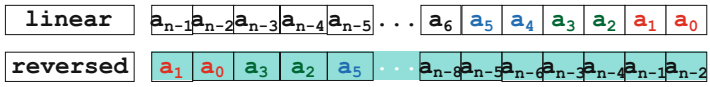


Fig. 39.16 Bit-level address mapping for output permutation in a 2^n-point in-order, permuted output radix-4 DIT FFT. $n = 2k$

Fig. 39.9a. The address mapping is obtained simply by reversing the bit-level representation of the address. For example, in Fig. 39.9a, the addressing sequence is $(0, 8, 4, 12, 2, \ldots, 15)$, which is obtained from the linear address in bit level as $0_{10} = 0000_2 \rightarrow 0_{10} = 0000_2$, $1_{10} = 0001_2 \rightarrow 8_{10} = 1000_2$, $2_{10} = 0010_2 \rightarrow 0_4 = 0100_2$, $3_{10} = 0011_2 \rightarrow 12_{10} = 1100_2$, etc. It should also be noted that the inverse permutation of bit reversal is the same bit reversal.

The transforms considered in this chapter, radix-4 and mixed-radix algorithms defined in Eqs. (39.7) and (39.14), respectively, contain output permutations, and the permutations are different than in radix-2 algorithms. The output reordering in 64-point radix-4 algorithm illustrated in Fig. 39.10 is listed in Fig. 39.16a. The bit-level representation shows that the 6-bit linear address is reversed in 2-bit fields to obtain the index of the permuted element. The general method for address mapping for output permutation in a 2^{2k}-point radix-4 DIT FFT is depicted in Fig. 39.16b; the address mapping is reversal of 2-bit fields in a $2k$-bit address.

The mixed-radix FFT has a different output permutation. The 32-point FFT in Fig. 39.11 has the output index sequence listed in Fig. 39.17a, which again shows the reversal of 2-bit fields. However, this time the address field contains an odd number of bits; thus the least significant bit is moved to the most significant bit. The general case for 2^{2k+1}-point FFT is depicted in Fig. 39.17b. By using the previous bit-level presentations, the complexity of address generation can be reduced significantly compared to using worldwide arithmetic operations.

a

Linear idx		Operand idx	
0	00000	0	00000
1	00001	16	10000
2	00010	4	00100
3	00011	20	10100
4	00100	8	01000
5	00101	24	11000
6	00110	12	01100
7	00111	28	11100
8	01000	1	00001
9	01001	17	10001
10	01010	5	00101
11	01011	21	10101
12	01100	9	01001
13	01101	25	11001
14	01110	13	01101
15	01111	29	11101
16	10000	2	00010

b

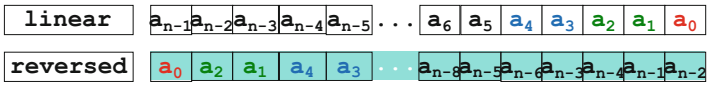


Fig. 39.17 Bit-level address mapping for output permutation in a 2^n-point in-order, permuted output mixed-radix DIT FFT. $n = 2k + 1$

39.5.3 Twiddle Factors

The twiddle factors defined in (39.4) are an integral part of FFT algorithms, and often these coefficients are stored in a look-up table and fetched during computation of the algorithms. While this is a simple and quick method for short transforms, the table size increases superlinearly with the transform size. Therefore, the coefficients are created at run time when working with longer transforms.

The twiddle factors are actually complex roots of unity evenly spaced in the unit circle on complex plane [7] as seen in Fig. 39.18. The number of different factors depends on the FFT size N and the type of the fast algorithm. For example, radix-2 algorithms contain $\frac{N}{2}\log_2 N$ twiddle factors, but there are only $\frac{N}{2}$ different factors. According to (39.10), in a radix-4 algorithm, there are three nontrivial twiddle factors in each butterfly, thus there is a total of $\frac{3}{4}N\log_4 N$ nontrivial twiddle factors while only $\left(\frac{N}{2}-1\right)$ are unique.

The twiddle factors can be formed by using trigonometric functions, which are, however, expensive. The coefficients can be computed with fast algorithms, which exploit the trigonometric identities of twiddle factors, e.g., Singleton's method [35]. Then the coefficients can be computed on the fly, when they are needed. The twiddle factors can be generated as piecewise polynomial approximation of a function. Polynomial approximation requires multiplications and additions to compute the value of a function with given parameters. It should also be noted that the complexity of the polynomial-based algorithm increases significantly with the required

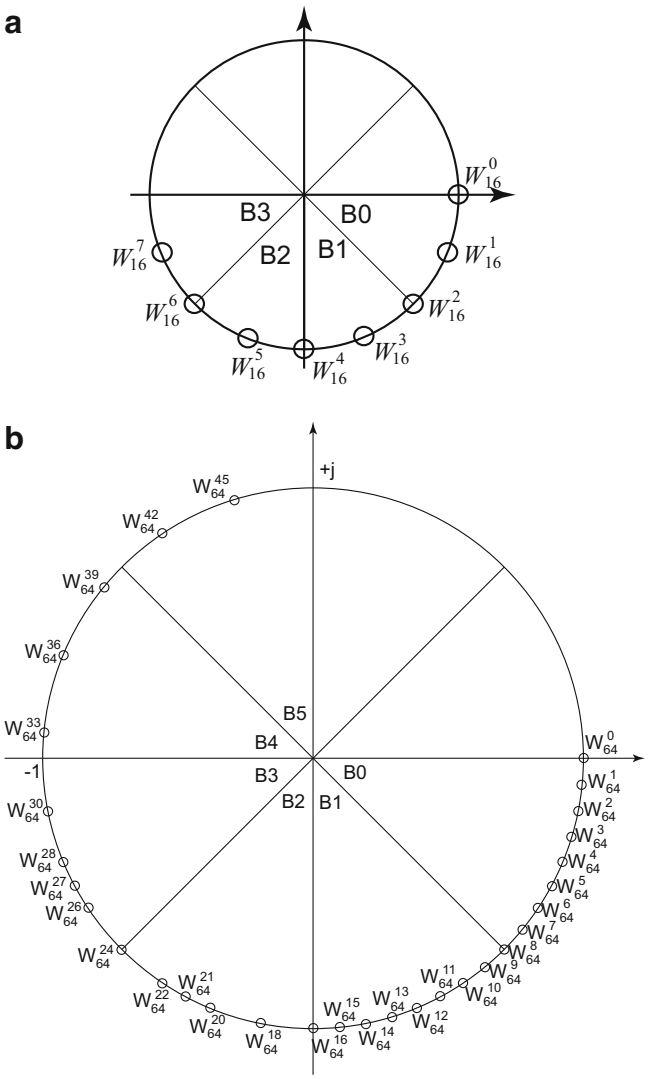


Fig. 39.18 Twiddle factors of (**a**) 16-point radix-2 and (**b**) 64-point radix-4 FFT in the complex plane

output precision. In [14], second-order polynomial approximation is combined with Horner's rule to compute the sine and cosine values.

Recursive twiddle factor generation is based on recursive feedback difference equations for sine and cosine functions. This approach is less complex compared to the polynomial, one iteration uses two real-valued multiplications and two real-valued additions to produce a complex-valued result. The drawback of the algorithm

is error propagation of the finite numbers due to the feedback structure of the algorithm. In [6], a method to reduce the complexity of error propagation circuit is proposed. The accuracy is improved with a correction table containing $\frac{N}{8}$ 3-bit entries. The area cost is reduced by sharing the same multiplier and adder for both real and imaginary parts, which doubles the latency. The method uses two look-up tables for cosine and sine values, and both tables require $\log_2 N - 2$ entries. The drawback is that the method generates an ordered sequence of twiddle factors; thus it supports only a specific type of FFT algorithms and the reported unit supports only radix-2 DIF FFT. In these algorithms, the large number of iterations will increase the length of computation kernel. This might increase the need for intermediate storage, i.e., registers. Also the large number of multiplications will increase the power consumption of twiddle factor generation.

Another method to compute the twiddle factors is to exploit the COordinate Rotational DIgital Computer (CORDIC) algorithm [13, 25, 44]. All of the trigonometric functions can be evaluated by rotating a unit vector in complex plane. This operation is effectively performed iteratively with the CORDIC algorithm. The general rotation transform at iteration t can be given as

$$\begin{cases} X_{t+1} = X_t \cos\phi - Y_t \sin\phi \\ Y_{t+1} = Y_t \cos\phi + X_t \sin\phi \end{cases}, \tag{39.17}$$

where (X_{t+1}, Y_{t+1}) is the resulting vector generated by rotation of an angle ϕ from the original vector (X_t, Y_t), i.e., the resulting vector rotates in the unit circle in similar fashion as the twiddle factors. Therefore, the CORDIC algorithm can be used to compute the twiddle factors, generating the sine and cosine values. In particular, CORDIC is used for replacing the twiddle factors with rotation information and, therefore, avoids multiplication with the twiddle factor by replacing it with rotation realized with additions.

The CORDIC multiplier consumes less power compared to a traditional multiplier. For example, in [47], a pipelined CORDIC unit consumed roughly 20% less power than the traditional complex-valued multiplier while the area cost was about the same. Recursive CORDIC iteration saves area compared to look-up-based twiddle factors, but it introduces longer latency. In [47], the rotation angle constants for generating all the twiddle factors for an N-point FFT are stored in a look-up table with $\log_2 N$ entries, while in [15], the twiddle factors are generated without pre-calculated coefficients. The CORDIC algorithm is iterative; thus it can be pipelined easily and it lends itself to pipelined FFT architectures. However, the dynamic power consumption with a large number of iterations and/or long pipeline will be higher than in a look-up table-based approach. This will be the case, when longer word widths are used, i.e., increased accuracy calls for more iterations. Traditionally the CORDIC has mainly been used in fixed-function ASICs, but it can be used to accelerate computations in a programmable processor as reported

in [34]. The authors describe instruction extensions for CORDIC operations, and there are separate instructions for vectoring and rotation mode.

Another approach is to exploit look-up tables and read the coefficients from the tables. In many cases, the look-up tables are stored in ROM, but, in software implementations, data memory is used to store the coefficients. The simplest design, radix-2, requires $\frac{N}{2}\log_2 N$ twiddle factors to be stored in the table. However, such a table contains redundancy as many of the coefficients are the same. In [27, 46], a method to reduce the number of coefficients in radix-2 algorithms to $\frac{N}{2}$ is proposed. Such a table can be used only for a sequential implementation, but, in [28], a method is proposed, which allows the $N/2$ entries to be distributed over 2^P, $P = 0, 1, \ldots, \log_2\left(\frac{N}{2}\right) - 1$ sub-tables such that those can be accessed by 2^P butterfly units simultaneously.

The previous twiddle factor table contains still redundancy: as the twiddle factors are equally spaced in unit circle on the complex plane, there is symmetry as illustrated in Fig. 39.18a. We can note that the real and imaginary parts of the twiddle factors in the octants B_0 and B_1 can be used to obtain the twiddle factors required in the octants B_2 and B_3. The number of look-up table entries in the radix-2 case can be reduced down to $\frac{N}{4} + 1$. Such an approach has been presented in [3, 30, 39]. In this method, the twiddle factors from the octants B_0 and B_1 in Fig. 39.18 are stored to the look-up table, and the rest of twiddle factors are generated simply by interchanging the real and imaginary parts of the coefficient and changing the sign according to the octant.

In the previous, the symmetry among different quadrants is exploited, but the symmetry between real and imaginary parts of the twiddle factors is not exploited. This would allow all the twiddle factors to be generated from only one octant, B_0 in Fig. 39.18a. In [17], method is shown, which avoids these redundancy twiddle factors for radix-2 FFTs are created with the aid of $\frac{N}{8} + 1$ complex-valued constants. In [32], this is extended to cover also radix-4 FFTs. The method can be used to construct twiddle factors for several transform sizes.

The redundancy in twiddle factors can easily be seen in Fig. 39.19; in order to represent all the different twiddle factors in 64-point radix-4 FFT, only the nine twiddle factors in the octant B0 are needed. For example, twiddle factor W_{64}^{14} in the octant B1 is obtained with the aid of W_{64}^{2} in octant B0: $W_{64}^{14} = -i\,W_{64}^{2\,*}$, where * denotes complex conjugate. In a similar fashion, $W_{64}^{18} = -i\,W_{64}^{2}$. In general case, for an N-point transform, we store the values from B0 to a table M:

$$M = (M_0, M_1, \ldots, M_{N/8}), \tag{39.18}$$

where an entry M_k in the table represents a twiddle factor W_N^k, which is computed based on the exponent k as follows:

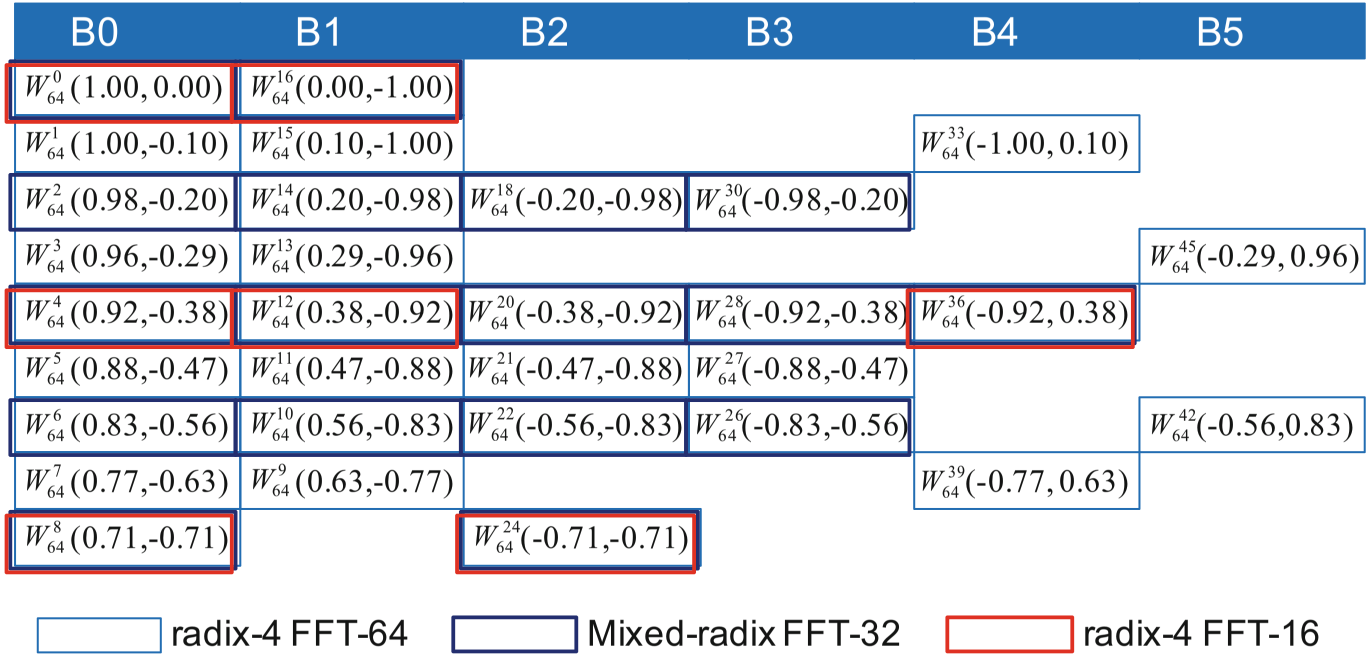


Fig. 39.19 Twiddle factors in 16-point radix-4 FFT, 32-point mixed-radix, and 64-point radix-4 FFTs in the different octants in complex plane in Fig. 39.18b

$$W_N^k = \begin{cases} M_A \text{ , when } 0 \le k \le \frac{N}{8} \\ -j\,M_A{}^* \text{ , when } \frac{N}{8} < k < \frac{N}{4} \\ -j\,M_A \text{ , when } \frac{N}{4} \le k \le \frac{3N}{8} \\ -M_A{}^* \text{ , when } \frac{3N}{8} < k < \frac{N}{2} \\ -M_A \text{ , when } \frac{N}{2} \le k \le \frac{5N}{8} \\ j\,M_A{}^* \text{ , when } \frac{5N}{8} < k \end{cases}, \tag{39.19}$$

where A is an index to the look-up table M obtained from the given exponent k. For N-point FFT, $N = 2^n$, k is represented with n bits; thus when using two's complement representation, the $(n-2)$-bit index A is obtained simply as

$$A = \begin{cases} k[n-3:0] \text{ , when } 0 \le k \le \frac{N}{8} \\ \sim k[n-3:0]+1 \text{ , when } \frac{N}{8} < k < \frac{N}{4} \\ k[n-3:0] \text{ , when } \frac{N}{4} \le k \le \frac{3N}{8} \\ \sim k[n-3:0]+1 \text{ , when } \frac{3N}{8} < k < \frac{N}{2} \\ k[n-3:0] \text{ , when } \frac{N}{2} \le k \le \frac{5N}{8} \\ \sim k[n-3:0]+1 \text{ , when } \frac{5N}{8} < k \end{cases}, \tag{39.20}$$

where $k[a:b]$ denotes the bit field $(k_a, k_{a-1}, \ldots, k_{b+1}, k_b)$ of a two's complement number $k = (k_{n-1}, \ldots, k_1, k_0)$ and $\sim$ is the bit-wise complement operation.

The look-up table can be used to create the twiddle factors in all the power-of-two FFTs smaller than N. This can be seen in Fig. 39.19: the twiddle factors in a 32-point mixed-radix FFT are a subset of twiddle factors in 64-point radix-4 FFT. The access to the table requires only a simple manipulation of parameter k as the twiddle factors in a 32-point FFT are every second twiddle factor in a 64-point FFT. In a similar fashion, the twiddle factors in a 16-point radix-4 FFT are a subset of twiddle factors in the 32-point FFT. A block diagram of a twiddle factor unit supporting all the power-of-two FFTs with 16-bit real and 16-bit imaginary precision is illustrated in Fig. 39.20. The actual twiddle factor generation requires only negation of sine and cosine values read from the look-up table as defined in (39.18). According to (39.20), the index to the look-up table is formed with simple operations: increment and complement, and modification of complex entries from the table uses only few simple gates and two full adders.

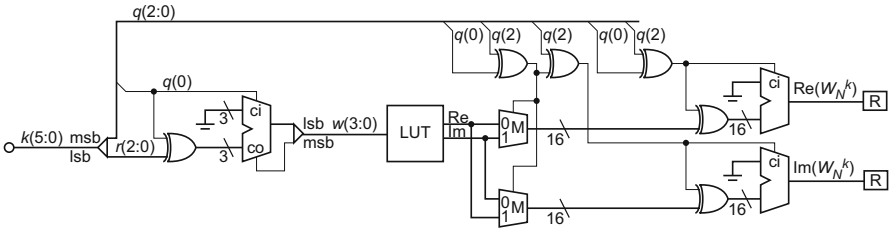


Fig. 39.20 Twiddle factor unit supporting all the power-of-two FFTs up to 64-points [32]

39.6 Customized FFT Architecture Based on Transport Triggering

The properties and optimizations discussed in the previous section can be used to tailor a single-thread transport-triggered processor for FFT computations. In this section, we describe the architecture, which is tailored by incorporating special function units discussed in the previous section.

The TTA template has been tailored according to the needs of radix-4 and mixed-radix FFT, and the resulting architecture can be seen in Fig. 39.21. The processor contains a set of standard function units, which has simply been taken from TCE hardware database, i.e., no design effort has been used to those units. The standard units include an instruction unit for controlling the operation, an immediate unit for extracting an immediate value from an instruction and passing it to the interconnection network, load/store units for accessing the data memories, a logical unit for standard logical operations, a comparator unit, and a shifter unit for arithmetic and logical shifts. There are several register files, which imply that the several temporary variables are accessed in the iteration kernel. There is one Boolean register for storing results of comparison, and this register can be used for predication to avoid costly branches.

The processor uses 32-bit arithmetic and packs a complex number in a single 32-bit word. There are 18 buses in the interconnection network; 17 are 32-bit buses (many of those are point-to-point buses), and there is a single 1-bit bus, which is used to transfer the Boolean results from comparisons. All these buses are generated by the ProGe tool once the processor architecture has been described with ProDe tool. The data memory is organized as a parallel memory consisting of two single-port memories with switching, which allows memories to be accessed through either of the two load/store units. This organization allows two memory accesses per clock

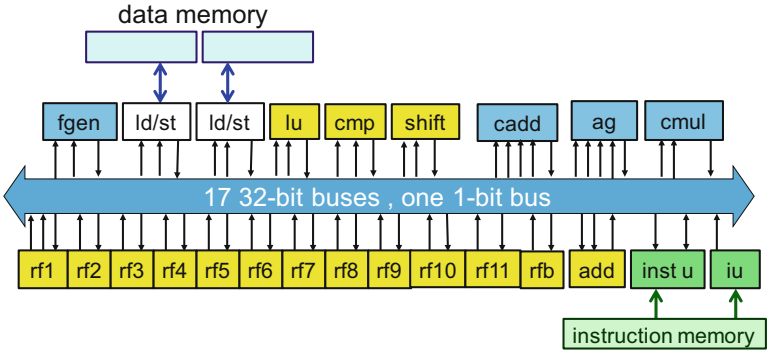


Fig. 39.21 Block diagram of TTA processor tailored for FFT computations. *fgen* twiddle factor generation unit, *ld/st* load/store unit, *lu* logical unit, *cmp* compare unit, *shift* shift unit, *cadd* complex-valued butterfly adder, *ag* operand address generation unit, *cmul* complex-valued multiplier, *rf* register file, *rfb* Boolean register, *add* adder unit, *inst u* instruction unit, *iu* immediate unit

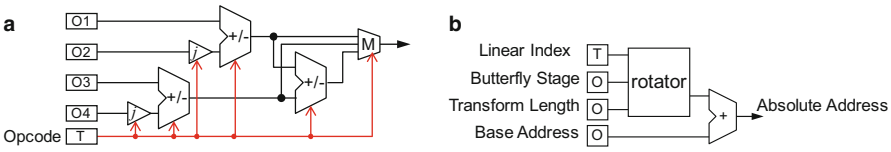


Fig. 39.22 Block diagram of (**a**) complex-valued butterfly adder and (**b**) operand address generator

cycle. The energy efficiency of the parallel memory is significantly better than the corresponding dual-port memory.

Finally, the processor has four special function units. The hardware structure of the units has been designed manually by exploiting the standard unit interface of the TTA template. There are separate units for complex-valued multiplication and complex-valued butterfly addition. The complex adder unit computes for different additions of four operands defined in 4-point DFT and two summations from 2-point DFT in (39.6). The block diagram of the complex butterfly unit is depicted in Fig. 39.22a. The unit has four operand ports and computes one of the outputs of butterfly operation when the opcode is transferred to the trigger port. The idea is that the four operands can be stored in the input registers over four consecutive clock cycles; thus there is no need to move operands, which reduces power consumption. The same unit can also be used to compute radix-2 butterflies when realizing mixed-radix FFTs.

The operand addresses are computed with a dedicated address generator unit illustrated in Fig. 39.22b. This is simply a rotator with an adder for adding the rotated index to the base address of the memory array. Once again, the linear index address is used as trigger port; thus during the FFT computations, all the other parameters are kept in input registers and operand moves can be avoided. If standard native arithmetic operations would be used for operand generation, it would take up to six operations. Here the customized unit can generate operand address at every clock cycle, which is sufficient to support two load/store units, as we perform in-place computations, i.e., the same address is used to read operand and store result.

The complex-valued twiddle factors are generated with a dedicated twiddle factor unit, which is based on the principle illustrated in Fig. 39.20. The unit can generate a new twiddle factor at rate of one per cycle. There are also some standard functional units for supporting control code. Many of the functional units are pipelined to support high clock frequency. The units have been designed such that during the FFT kernel computations, the throughput is one operation per clock cycle.

The programmability of the processor is usually limited if heavy customizations are used, i.e., when general-purpose functional units are removed. However, this architecture is still programmable, but the performance in general applications is limited. The TCE tools can be used to compile code on the customized architecture.

The code for FFT application is developed by exploiting heavily software pipelining and loop unrolling. Figure 39.23 shows the reservation table of the 17 buses during the computation of radix-4 FFT. Each color in the figure denotes move

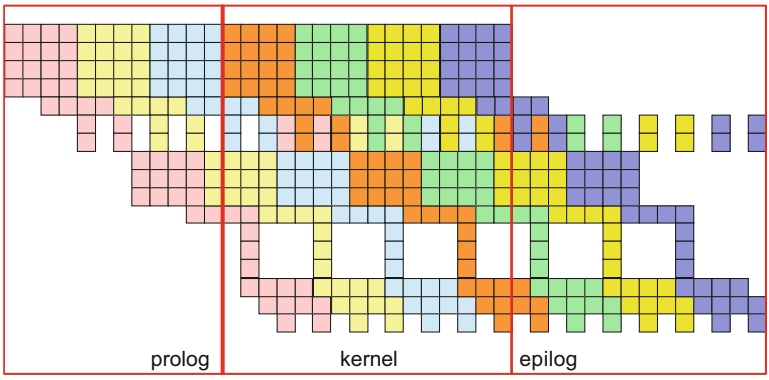


Fig. 39.23 Reservation table for the radix-4 FFT code

instructions related to computation of a single radix-4 butterfly with twiddle factors. The figure shows that the many of the resources are fully reserved during the kernel computation. The actual computation kernel contains 16 instructions while it could have been also shorter. However, the intermediate results are stored in register files over few clock cycles; thus the shorter kernel would require that the intermediate values would be stored in a FIFO type of storage. This would need data to be moved from register to register every cycle, which would consume extra power. Therefore, the intermediate values are kept in registers and kernel code need to be accessed from memory. However, in this specific case, we exploited code compression [19]; thus the instruction word is short.

Two versions of processors are developed: one with larger memory for supporting all the power-of-two FFT up to 16k-points and another version with smaller memory to support only 1-point FFT. Both the processors have the same processor core, but data memory and software are different.

Both the processors can perform two memory accesses per cycle; thus the overhead of the system can be compared by determining the theoretical lower bound for the number of memory cycles required for computing 1024-point radix-4 FFT. The radix-4 computation requires $1024 \log_4(1024)$ memory reads and memory writes; thus for a two-port memory, a total of 5121 memory accesses are needed. In the customized TTA core, computation of 1k-point FFT takes 5208 cycles; thus it shows really low overhead. The processors have been synthesized on a 130nm IC technology, and analysis shows maximum clock frequency of 250 MHz at 1.5 V supply voltage and 140 MHz at 1.1 V. The processor core takes 33 kgates, and in the smaller memory configuration for 1k-point FFT, memory is 30 kgates, while the larger memory supporting 16k-point FFT takes 240 kgates. The total power consumption for 1k-point FFT is 59 mW@250 MHz with 1.5 V supply voltage, where the smaller memory uses 16 mW. The most power hungry unit in the core is the twiddle factor generator, which takes about 23% of the core area and 7% of the power consumption. When several transform sizes are supported with the larger

memory, the power consumption is higher when mixed-radix code is executed, e.g., computing

The major effort in the actual design work was spent on finding out the specific features of the algorithm, which can be exploited to speed up the computations. The effect of candidate features was analyzed by creating a high-abstraction-level model of the function unit, which was then used in simulator. Once the unit was verified to be useful, only then RTL code for the unit was developed; thus there was a need to develop RTL code only for four units. The RTL for the processor was generated with ProGe tool, and the design was synthesized with commercial IC design tools.

39.7 Energy Efficiency Comparison

Power consumption is a usual design metric when designing energy-efficient systems. However, power consumption depends on several issues: computing resources, memories, caches, computation cycles, operating voltage, and operation frequency. The energy efficiency can be compared by measuring the energy consumed for performing a reference task. Here we compare the energy efficiency by measuring how many 1024-point FFTs can be computed with energy of 1 mJ. This approach tries to compensate the effect of computational speed, but there are other implementation-specific parameters, which have a great effect on the result.

There are still some parameters, which may differ, such the implementations should be normalized. Although exact scaling of the characteristics of an implementation on a specific IC technology to another technology is difficult, even impossible, there are several normalization methods proposed in the literature. A normalization method for IC technologies is proposed in [38], which tries to take into account many architectural aspects and implementation-specific features; the normalized energy consumption of a system, E_N, is defined as

$$E_N = E \frac{L_r U_r^2 \left(\frac{1}{3} W_r^2 + \frac{2}{3} W_r\right)}{L U^2 \left(\frac{1}{3} W^2 + \frac{2}{3} W\right)}, \tag{39.21}$$

where E is the energy consumption of the system implemented on a specific IC technology, W is the word length of the system, U is the supply voltage of the implementation, and L is feature size of the specific IC technology on which the system has been implemented. The energy consumption of the implemented system is normalized for the same system implemented on reference technology, where L_r is the feature size of the reference IC technology, U_r is the supply voltage of reference technology, and W_r is the word length of the reference design.

We have compared the energy efficiency of the developed TTA processor against several other FFT implementations by using the previous normalization. The following parameter set has been used: $L_f = 130\,\text{nm}$, $U_r = 1.5\,\text{V}$, and $W_r = 16$ bits.

The energy efficiency comparisons are shown in Table 39.1. As expected FFT implementation on a general-purpose processor [5] has low energy efficiency.

Table 39.1 Energy efficiency comparison of various normalized FFT implementations measured as the number of executed 1024-point FFTs with energy of 1 mJ.

Design	Tech.	Class	WL	V_{CC}	t_{clk}	t_{FFT}	Efficiency	
	[nm]		[bits]	[V]	[MHz]	[μs]	[FFT/mJ]	
[5]	65	GPP	16	1.2	1000	63	1	†
[41]	130	DSP	16	1.5	720	8	100	†
[37]	45	ASIC	32	0.9	650	2	1007	
[4]	180	ASIC	13	1.8	51	61	748	
[38]	180	ASIC	14	1.8	5	220	755	
[45]	65	ASP	16	1.2	150	6	633	†
[16]	130	ASP	16	1.5	320	14	1170	†
[1]	180	ASP	16	1.8	280	37	61	†
TTA [33]	130	ASP	16	1.5	250	21	809	

V_{CC} Supply voltage, *WL* Word length, t_{clk} Clock period, t_{FFT} FFT execution time, *GPP* General-purpose processor, *DSP* Digital signal processor, *ASIC* Application-specific integrated circuit, *ASP* Application-specific processor

† Energy does not include memories.

The Digital Signal Processor (DSP) in [41] can achieve high performance, but the energy efficiency in high-speed mode is lower than in low-power mode, i.e., lower frequency and supply voltage. In addition, the high performance calls for manually optimized assembly code. It should be noted that the energy figures exclude memories.

The application-specific processor in [1] contains user-specific function units, e.g., for address generation and butterfly computations. There are two complex-valued multipliers and three complex-valued adders. The twiddle factors are stored in the main memory. The pipeline architecture in [37] realizes data permutation with the aid of delay lines, where data traverses through the registers introducing high dynamic power consumption. It is not known if the twiddle factor memories and address generators are included in the energy figures. Another pipelined processor is proposed in [38], which uses block floating-point number representation with 10-bit mantissa and 4-bit shared exponent. The short floating-point word allows CORDIC pipeline to be shortened. If larger word lengths are needed, e.g., to support larger FFTs or to improve the signal-to-noise ratio, the pipeline depth needs to be increased, which increases the power consumption.

A cache-memory architecture is described in [4], where a small data cache is used to reduce accesses to the main memory. The processor uses 13-bit complex data type and supports FFT size up to 1024-points. In [45], a small cache memory is also used. The twiddle factors are stored in the main memory, which adds power consumption. Unfortunately, caches or memories are excluded from the energy figures.

The application-specific processor in [16] has two small caches to reduce access to the main memory, and these are accessed in ping-pong fashion to avoid stall cycles when transferring the results to the main memory. The processor uses $\left(\frac{N}{8}+1\right)$ complex-valued coefficients to compute the twiddle factors. External data

memories are not included in the energy figures. In addition, the power consumption figures are coarse estimates obtained from a processor design tool.

39.8 Conclusions

In this chapter, we described transport-triggered architecture template, which can be used to develop application-specific processors. In addition, we introduced the TCE hardware/software codesign environment for developing tailored implementations based on TTA processors. The TCE provides tool support for iterative processor customization starting from high-level programming languages and contains retargetable compiler, which speeds up the iterative customization significantly. The tools produce synthesizable RTL description of the TTA processor and generates instruction parallel binary code. TCE is available as a liberally licensed open-source project and can be downloaded from the web page [40]. We also customized a TTA processor for FFT application and showed that the highly customized but still programmable processor possesses energy efficiency close to fixed-function ASIC implementations.

Acknowledgments The authors thank the Finnish Funding Agency for Innovation in the context of the FiDiPro project StreamPro (decision no. 40142/14).

References

1. Baek JH, Kim SD, Sunwoo MH (2008) SPOCS: application specific signal processor for OFDM communication systems. J Signal Process Syst 53(3):383–397. doi: 10.1007/s11265-008-0240-4
2. Chang WH, Nguyen TQ (2008) On the fixed-point accuracy analysis of FFT algorithms. IEEE Trans Signal Proc 56(10):4673–4682
3. Chang YN, Parhi KK (1999) Efficient FFT implementation using digit-serial arithmetic. In: Proceedings of IEEE international workshop signal processing system, Taipei, pp 645–653. doi: 10.1109/SIPS.1999.822371
4. Chen CM, Hung CC, Huang YH (2010) An energy-efficient partial FFT processor for the OFDMA communication system. IEEE Trans Circuits Syst II 57(2):136–140. doi: 10.1109/TCSII.2010.2040318
5. Cheng KT, Wang YC (2011) Using mobile GPU for general-purpose computing: a case study of face recognition on smartphones. In: Proceedings of international symposium VLSI design automation test, Hsinchu, pp 1–4. doi: 10.1109/VDAT.2011.5783575
6. Chi JC, Chen SG (2004) An efficient FFT twiddle factor generator. In: Proceeding of European signal processing conference, Vienna, pp 1533–1536
7. Chu E, George, A (2000) Inside the FFT black box: serial and parallel fast Fourier transform algorithms. CRC Press, Boca Raton
8. Cichon G, Robelly P, Seidel H, Matúš E, Bronzel M, Fettweis G (2004) Synchronous transfer architecture (STA). In: Computer systems: architectures, modeling, and simulation. Lecture notes in computer science, vol 3133. Springer, Berlin/Heidelberg, pp 193–207. doi: 10.1007/978-3-540-27776-7_36
9. Cooley JW, Tukey JW (1965) An algorithm for the machine calculation of complex Fourier series. Math Comput 19(90):297–301

10. Corporaal H (1997) Microprocessor architectures: from VLIW to TTA. Wiley, Chichester
11. Corporaal H, Mulder H (1991) MOVE: a framework for high-performance processor design. In: Proceedings of ACM/IEEE conference on supercomputing, Albuquerque, pp 692–701. doi: 10.1145/125826.126159
12. Dally W, Balfour J, Black-Shaffer D, Chen J, Harting R, Parikh V, Park J, Sheffield D (2008) Efficient embedded computing. Computer 41:27–32. doi: 10.1109/MC.2008.224
13. Despain AM (1974) Fourier transform computers using CORDIC iterations. IEEE Trans Comput C-23(10):993–1001. doi: 10.1109/T-C.1974.223800
14. Fanucci L, Roncella R, Saletti R (2001) A sine wave digital synthesizer based on a quadratic approximation. In: Proceedings of IEEE international frequency control symposium PDA exhibition, pp 806–810. doi: 10.1109/FREQ.2001.956385
15. Garrido M, Grajal J (2007) Efficient memoryless CORDIC for FFT computation. In: Proceedings of IEEE international conference acoustics speech signal processing, Honolulu, vol 2, pp 113–116. doi: 10.1109/ICASSP.2007.366185
16. Guan X, Fei Y, Lin H (2012) Hierarchical design of an application-specific instruction set processor for high-throughput and scalable FFT processing. IEEE Trans VLSI Syst 20(3): 551–563. doi: 10.1109/TVLSI.2011.2105512
17. Hasan M, Arslan T (2002) FFT coefficient memory reduction technique for OFDM applications. In: IEEE international conference acoustics speech signal process, Orlando, vol 1, pp 1085–1088
18. He Y, She D, Mesman B, Corporaal H (2011) MOVE-Pro: a low power and high code density TTA architecture. In: Proceedings of international conference on embedded computer system: architectures modeling simulation, pp 294–301. doi: 10.1109/SAMOS.2011.6045474
19. Heikkinen J, Takala J, Corporaal H (2009) Dictionary-based program compression on customizable processor architectures. Microprocess Microsyst 33(2):139–153. doi: 10.1016/j.micpro.2008.10.001
20. IEEE 802.16.1 (2012) IEEE standard for wireless MAN – advanced air interface for broadband wireless access systems. Std 802.16.1–2012. IEEE
21. Jääskeläinen P, Kultala H, Viitanen T, Takala J (2014) Code density and energy efficiency of exposed datapath architectures. J Signal Process Syst 1–16. doi: 10.1007/s11265-014-0924-x
22. Jääskeläinen P, de La Lama C, Huerta P, Takala J (2011) OpenCL-based design methodology for application-specific processors. Transactions on HiPEAC 5. Available online
23. Jääskeläinen P, de La Lama CS, Schnetter E, Raiskila K, Takala J, Berg H (2014) pocl: a performance-portable OpenCL implementation. Int J Parallel Prog 1–34. doi: 10.1007/s10766-014-0320-y
24. Jääskeläinen P, Salminen E, de La Lama C, Takala J, Ignacio Martinez J (2011) TCEMC: a co-design flow for application-specific multicores. In: Proceeding of international conference on embedded computer system: architectures modeling and simulations, Samos, pp 85–92. doi: 10.1109/SAMOS.2011.6045448
25. Jiang RM (2007) An area-efficient FFT architecture for OFDM digital video broadcasting. IEEE Trans Consum Electron 53(4):1322–1326. doi: 10.1109/TCE.2007.4429219
26. Johnson H, Burrus C (1984) An in-order, in-place radix-2 FFT. In: IEEE international conference on acoustics speech signal processing, vol 9, San Diego, pp 473–476. doi: 10.1109/ICASSP.1984.1172660
27. Johnsson SL, Krawitz RL, Frye R, MacDonald D (1989) A radix-2 FFT on connection machine. In: Proceeding of ACM/IEEE conference on supercomputing, Reno, pp 809–819. doi: 10.1145/76263.76355
28. Jui PC, Wey CL, Shiue MT (2013) Low-cost parallel FFT processors with conflict-free ROM-based twiddle factor generator for DVB-T2 applications. In: Proceedings of IEEE international midwest symposium circuits system, Columbus, pp 1003–1006
29. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 international symposium on code generation and optimization (CGO'04), Palo Alto

30. Ma Y, Wanhammar L (2000) A hardware efficient control of memory addressing for high-performance FFT processors. IEEE Trans Signal Process 48(3):917–921. doi: 10.1109/78.824693
31. Oppenheim AV, Schafer RW (2010) Discrete-time signal processing, 3rd edn. Pearson, Upper Saddle River
32. Pitkänen T, Partanen T, Takala J (2007) Low-power twiddle factor unit for FFT computation. In: Vassiliadis S, Berekovic M, Hämäläinen T (eds) Embedded computer systems: architectures, modeling, and simulation. Proceeding of 7th international workshop SAMOS VII, vol LNCS 4599. Springer, Berlin, pp 233–240. doi: 10.1007/978-3-540-73625-7_9
33. Pitkänen T, Takala J (2011) Low-power application-specific processor for FFT computations. J Signal Process Syst 63(1):165–176. doi: 10.1007/s11265-010-0528-z
34. Senthilvelan M, Sima M, Iancu D, Schulte M, Glossner J (2013) Instruction set extensions for matrix decompositions on software defined radio architectures. J Signal Process Syst 70:289–303. doi: 10.1007/s11265-012-0665-7
35. Singleton R (1967) A method for computing the fast Fourier transform with auxiliary memory and limited high-speed memory. IEEE Trans Audio Electroacoust 15(2):91–98
36. Strang G (1994) Wavelets. Am Sci 82(3):250–255
37. Suleiman A, Saleh H, Hussein A, Akopian D (2008) A family of scalable FFT architectures and an implementation of 1024-point radix-2 FFT for real-time communications. In: IEEE international conference on computer design, Lake Tahoe, pp 321–327. doi: 10.1109/ICCD.2008.4751880
38. Tang SN, Liao CH, Chang TY (2012) An area- and energy-efficient multimode FFT processor for WPAN/WLAN/WMAN systems. IEEE J Solid-State Circuits 47(6):1419–1435. doi: 10.1109/JSSC.2012.2187406
39. Tang Y, Qian L, Wang Y, Savaria Y (2003) A new memory reference reduction method for FFT implementation on DSP. In: Proceedings of ISCAS, Bangkok, vol 4, pp 496–499. doi: 10.1109/ISCAS.2003.1205932
40. TTA-based co-design environment (2015). http://tce.cs.tut.fi. Accessed: 15 Jan 2016
41. Texas Instruments, Inc. (2003) TMS320C64x DSP Library programmer's reference, Dallas
42. Thuresson M, Själander M, Björk M, Svensson L, Larsson-Edefors P, Stenström P (2007) FlexCore: utilizing exposed datapath control for efficient computing. In: Proceedings of international conference on embedded computer system: architectures modeling simulation, Samos, pp 18–25. doi: 10.1109/ICSAMOS.2007.4285729
43. Viitanen T, Kultala H, Jääskeläinen P, Takala J (2014) Heuristics for greedy transport triggered architecture interconnect exploration. In: Proceedings of international conference compilers architecture synthesis embedded system, New Delhi, pp 2:1–2:7. doi: 10.1145/2656106.2656123
44. Volder JE (1959) The CORDIC trigonometric computing technique. IRE Trans Electron Comput EC–8(3):330–334. doi: 10.1109/TEC.1959.5222693
45. Wang W, Li L, Zhang G, Liu D, Qiu J (2011) An application specific instruction set processor optimized for FFT. In: IEEE international midwest symposium circuits and systems, Seoul, pp 1–4. doi: 10.1109/MWSCAS.2011.6026391
46. Wanhammar L (1999) DSP integrated circuits. Academic Press, San Diego
47. Yu CY, Chen SG, Chih JC (2006) Efficient CORDIC designs for multi-mode OFDM FFT. In: Proceedings IEEE international conference acoustics speech signal processing, vol 3, Toulouse, pp III-1036–III-1039. doi: 10.1109/ICASSP.2006.1660834

Embedded Computer Vision

40

Marilyn Wolf

Abstract

Embedded computer vision is a challenging application domain, requiring high computation rates, high memory bandwidth, and support for a wide range of algorithms. This chapter reviews basic concepts in computer vision, design methodologies for embedded computer vision, platform architectures, and application-specific architectures.

Acronyms

CGA	Coarse-Grained Array
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
DRAM	Dynamic Random-Access Memory
FPGA	Field-Programmable Gate Array
GOPS	Giga Operations Per Second
GPU	Graphics Processing Unit
HSCD	Hardware/Software Codesign
MAC	Multiply-Accumulator
MPSoC	Multi-Processor System-on-Chip
NoC	Network-on-Chip
QoS	Quality of Service
RC	Reconfigurable Cell
RISC	Reduced Instruction-Set Processor
SoC	System-on-Chip
SPI	Signal Passing Interface
VLIW	Very Long Instruction Word

M. Wolf (✉)
School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA
e-mail: wolf@ece.gatech.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_40

Contents

40.1 Introduction

Embedded Computer Vision (embedded CV) is well-suited to Hardware/Software Codesign (HSCD). Computer vision is highly compute intensive and memory intensive. A diverse set of algorithms is used to build computer vision applications. And computer vision is often required to perform in real-time and on limited power budgets. Thanks to all these requirements, codesign is an important approach for the design of embedded computing systems.

Design methodologies and architectures are closely coupled in this domain. The high bandwidth and compute performance required by vision often pushes us to nonstandard architectures. Different architectural models require different types of synthesis algorithms.

Embedded computer vision is well-suited to HSCD and also presents significant challenges. The high-throughput and low-power consumption required for embedded CV make it a natural candidate for the customized solutions offered by HW/SW codesign. However, those intensive computational requirements also make it difficult to apply totally automated approaches to the design of embedded CV systems. Any design must address several important problems simultaneously: memory system organization, low power, and numerical accuracy. Not only do most codesign systems address only some of these issues but managing the trade-offs between these aspects is inherently challenging. HSCD is often used in two ways: to explore the design space and identify good hardware and software architectures and to design subsystems that can be integrated into larger systems.

We will start with a review of concepts in computer vision. We will then consider work on methodologies for the design of computer vision systems. We will next evaluate the range of architectures proposed for computer vision systems. We will then look at application-specific solutions to various problems in computer vision. We then compare approaches followed by a discussion of practical codesign methodologies for embedded computer vision.

40.2 Computer Vision Concepts

Computer vision applications can be roughly grouped into several categories:

- *Detection* categorizes classes of objects but not the identity within that class – for example, a generic vehicle but not its make or model.
- *Recognition* identifies specific types of objects, such as a type of vehicle.
- *Tracking* determines the movement of an object over time.

Computer vision applications can often be divided into several phases:

- *Feature extraction* performs relatively local operations to identify features in the image. Static features in a single frame include lines, curves, and textures. Analysis of motion provides features over a set of frames. Background elimination, in which pixels are classified as either foreground (motion exhibiting certain characteristics) or background (other pixels), is another form of feature extraction.
- *Classification* identifies sets of features as belonging to a category of interest: faces, vehicles, etc.

Although the human visual cortex can perform many different types of complex vision tasks, computer vision research has not yet identified a unified model for vision algorithms. Different vision applications have different requirements, and different types of algorithms are often used to solve those problems. Many vision systems are complex pipelines with multiple stages of processing. The algorithms at different stages of the pipeline can vary significantly, as can vary their required data rates. Early stages generally consume the most bandwidth, while later stages may use more abstract, compact representations. Algorithmic kernels at each stage may also contain a significant amount of control – filtering is not an adequate computational model for computer vision kernels. However, we can identify a stable of useful computational kernels that can be applied in different combinations to a wide range of vision applications.

40.3 Methodologies

Higher-level programming models have been very successful in the signal processing domain. Models such as data flow are well-matched to algorithmic expression and allow compilers to compile operator and data schedules, buffer sizes, and other program characteristics.

Bhattacharyya and colleagues have developed an extensive methodology around advanced versions of synchronous data flows; they have also conducted several demonstrations on computer vision applications. The signal passing interface [20] (SPI) extends synchronous dataflow models for varying rates of data transfer.

A variable data rate with an upper bound on the rate is modeled as a virtual token; conditions are developed under which this model may be converted to a traditional synchronous dataflow model for which schedules and buffer sizes can be found. Kianzad et al. [5] used the synchronous dataflow design-flow model to design a face detection accelerator. A face is modeled as an ellipse; a double exponential filter is used to detect the ellipse. Saha et al. [21] designed an architecture for particle filters. A particle filter module performs three steps: a set of particles (samples) are generated using a sampling function; an importance weight is calculated for each particle; the set of particles is resampled. A module includes a core processor, noise generator, weight update unit, and memory interfaces.

Networks on chips have received some attention for application-specific design given the high bandwidth and heterogeneous characteristics of vision applications. Xu et al. [33] developed a design methodology for application-specific networks on chips. They start by analyzing the communication patterns between the processing elements to determine communication rates, frequencies, and volumes. They use a recursive algorithm to design a hierarchical network topology with switches mediating between units in a cluster. They also generate a packet format. They then estimate the NoC floor plan to give both areas and link delays. They then use a network simulator to perform a detailed analysis of network operation. The final step is to analyze network performance and area based on a component library.

Numerical accuracy is an important concern in all aspects of digital signal processing as well as computer vision. Choosing the numerical representation appropriate to each algorithmic stage reduces hardware requirements and allows more computational units to be placed on chip. Schlessman and Wolf [22] describe a methodology for the analysis of trade-offs between numerical accuracy, performance, and power in computer vision modules. They applied their methodology first to the Kanade-Lucas-Tomasi optical flow algorithm [15]. Their analysis showed that the 18×18 integer multipliers available on the target FPGA did not provide enough dynamic range. They instead used a 24-bit floating-point format with a six-bit exponent. They then applied their methodology to mixture-of-Gaussians background elimination [10], for which they were able to eliminate all floating-point operations and substantially reduce memory requirements as compared to direct synthesis from Matlab code. Their cost analysis for floating-point operations was based on the work of Soderquist and Leeser [23].

40.4 Platform Architectures

A number of platform architectures are used or have been proposed for computer vision. Given the high-performance requirements for computer vision, these platforms tend to be significantly different from those used for IT-oriented workloads. We will first look at multiprocessor system-on-chip architectures and then networks on chips and finally Field-Programmable Gate Arrays (FPGAs) and coarse-grained arrays.

40.4.1 Multiprocessor Systems on Chips

A Multi-Processor System-on-Chip (MPSoC) is a single chip that contains multiple complex processing elements as well as some amount of on-chip memory and off-chip memory control. Wolf et al. [31] survey the development of MPSoCs. Many video-oriented chips are heterogeneous MPSoCs that combine several types of processors. MPSoCs are widely used for commercial embedded computing systems.

A well-known example of a traditional MPSoC for multimedia and computer vision is the Texas Instruments TMS320DM816x DaVinci family [11]. This architecture combines a Central Processing Unit (CPU), a Very Long Instruction Word (VLIW), and accelerators. An ARM Cortex-A8 provides in-order dual-issue processing and NEON multimedia instructions. The TMS320C674x VLIW has six function units, two multiply units, and 46 general-purpose registers. Accelerators provide operations for video encoding and decoding.

The Mobileye's EyeQ processor [24] is designed for automotive computer vision. It includes two ARM CPUs, accelerators for image scaling, tracking, lane detection, and image filtering.

40.4.2 Networks on Chips

Networks-on-Chips (NoCs) are widely used in cellphone processors to support multimedia data. NoCs often have application-specific topologies with varying bandwidths in different parts of the system. They also use smart adapters that directly handle packet scheduling without the intervention of the host processor. van der Wolf and Geuzebroek [26] analyzed Quality of Service (QoS) mechanisms; they identified key parameters as bandwidth, latency, number of transactions per burst, time between bursts, maximum number of pending transactions, and ratio of read to write traffic. Weber et al. [29] designed an epoch-based scheduling algorithm for NoCs that takes into account both the QoS requirements and the network state in that epoch.

Xu et al. [32] studied the design space of NoC architectures for smart cameras. They used as their test case the gesture recognition system of Wolf et al. [30]. They compared bus and crossbar architectures for the switches. They evaluated several different switch configurations with port widths ranging from 5 to 55 bits. They also evaluated a crossbar design with two shared memory: four processing elements share one of the memories, while the other three PEs share the other. They used the OPNET network simulator to evaluate network behavior using traces generated from the smart camera application. They judged the designs based on utilization and the required network frequency to support 150 frames/sec operation. Their results showed a wide variation in network characteristics for the various NoC architectures. A 16-bit/port design with a 3×3 crossbar and two memories gave 85.7% higher performance than the baseline processor-controlled network and required only a 443 MHz clock frequency to achieve the required frame rate. This

network had the lowest maximum throughput, which results in lower worst-case hardware requirements and a smaller-area implementation.

40.4.3 FPGAs and Coarse-Grained Arrays

FPGAs have been used as the fabrics for many computer vision systems. Gudis et al.[9] use a crossbar-connected set of processing elements and a video DMA controller. Farabet et al. [8] use a dataflow-oriented architectural style.

Coarse-Grained Arrays (CGAs) have been developed by a number of groups to provide more efficient implementations of certain classes of applications. As compared to FPGAs, a larger block of logic is predesigned to reduce the overhead of reconfigurability; these coarse-grained units often include some amount of memory. Like FPGAs, many of them have separate configuration and execution modes. While a multi-core processor may route packets dynamically, with sources and destinations varying continually, CGAs generally commit interconnection resources at configuration time.

MorphoSys [14] uses reconfigurable array controlled by a Reduced Instruction-Set Processor (RISC) processor. The control processor initiates all data transfers and generates control signals for the reconfigurable array processing elements. A processing element includes an ALU, a multiplier, and register files.

The ADRES architecture [3] uses a coarse-grid array controlled by a VLIW-style control unit. The control unit array is a two-dimensional mesh processing elements. Each processing element includes a function unit and a register file. The processing elements also share global data and program register files. The VLIW control starts and stops loops to be executed on the coarse-grid array. A suite of compilation tools can be used to map application code onto ADRES.

The MORA architecture [13] is designed for a large number of data operations and distributed internal memory with large internal bandwidth. A Reconfigurable Cell (RC) includes a data path, a RAM, and a control unit. Connections between the RCs are determined statically at configuration time; routing resources allow connections to vertical, horizontal, and diagonal neighbors. A memory controller feeds the RCs from internal controller.

Park et al. [19] proposed a polymorphic pipeline array for multimedia processing. A core consists of a set of four processing elements each with a function unit and register file, a loop buffer, and an instruction cache. The PEs in a core are connected with a mesh network. Cores are also interconnected using a mesh network. Vertical connections through the mesh connect several columns of cores to a memory arbiter.

Tabhki et al. [25] propose a function-level processor architecture for video and computer vision. A set of function blocks perform data operations. A function block typically performs operations at the granularity of computational kernels such as convolution. The function block can operate either under the control of a CPU or using its own program. Each function block includes registers for configuration and control and a buffer for parameters that can be written by other blocks. Data is fed into and out of the function block using streaming controllers. Parameter data is

cached while streaming data is not. Function blocks communicate with each other using a sparse multiplexer network.

The EGRA architecture [2] makes use of heterogeneous reconfigurable ALU clusters. These clusters may contain several different types of ALUs and can be chained together combinationally to operate on large expressions.

Networks on chips are widely used in cellphone processors to handle quality-of-service (QoS)-oriented multimedia traffic [26, 27, 29]. These networks generally allow prioritization and QoS parameterization. Flow control is handled directly by the network adapters, not by the host processors.

40.5 Application-Specific Architectural Solutions

A great deal of work has concentrated on accelerators for particular applications or computational kernels. We will consider foreground/background detection and then face detection and recognition. We will conclude with a survey of architectures for convolutional neural networks.

40.5.1 Foreground Detection

Many computer vision systems use motion to segregate pixels at an early stage, a process known as foreground detection. In tracking, for example, pixels with only a small amount of motion are less likely to be part of an object of interest. Foreground/background analysis classifies pixels: foreground pixels have a significant amount of interesting motion, while background pixels do not.

Casares et al. [4] developed a foreground detection algorithm with a small memory footprint. Each pixel is given a binary-valued state $s(i, j)$. A counter for each pixel location keeps track of the number of times the pixel's state has changed in the last 100 frames. An image is classified as foreground only if its change count falls below a threshold. A pixel may be added to the foreground model with partial weight based upon its activity counter.

40.5.2 Face Detection and Recognition

Face detection and recognition are both useful computer vision operations. Face detection determines the location of a human face in the scene, while face recognition identifies the person associated with that face.

Amir et al. [1] developed a hardware design for a real-time eye tracker. Their algorithm makes use of synchronized illuminators, one on axis and the other off axis. Pupils appear much brighter with on-axis illumination than with off-axis illumination. The algorithm captures a pair of frames, one with each type of illumination, and then subtracts the off-axis illuminated image from the on-axis illuminated frame. They generate a binary image of the difference by thresholding

the pixel values and then use a connected components algorithm to identify regions. Their connected components algorithm operates on pairs of lines; they identify line segments in the two lines by comparing the start and end points of the line segments. Region identifications are maintained vertically to generate consistent component labels. As regions are merged, the moment of the newly enlarged region is created. Moments are used to identify possible pupils; the centroid of each identified region is also computed.

EFFEX [7] is an accelerator for feature extraction algorithms. It includes a one-to-16 comparison unit, a convolution Multiply-Accumulator (MAC), and a gradient unit. By storing data in the off-chip Dynamic Random-Access Memory (DRAM) in the proper order, a patch of data can be read by accessing a single DRAM row.

Yang et al. [34] developed an AdaBoost-based algorithm for face detection. The AdaBoost approach to face detection was originally proposed by Viola and Jones [28]. Their algorithm makes use of Haar-like features that they represent in the form of integral images. The basic Haar-like feature is defined on a 2×2 set of pixels, although it can also be defined on larger image patches. Figure 40.1 shows the four regions of a 2×2 patch. The value of each region is the sum of the pixels in the fourth quadrant relative to that region (above and to the left). The sum of any rectangular area in the image is computed as $S = C + A - B - D$. These features can be applied at any image scale. Their face detection algorithm detects Haar-like features at multiple image resolutions. At each stage, a classifier is composed of the response of a set of feature responses. The classifiers are tested at multiple locations with classifiers that pass the threshold test being passed onto the next stage of image resolution. The highest-valued classifier is selected as the face location in the final stage. Yang et al. developed heuristics to control the number of features propagating through the architecture in order to ensure that face detection time remained constant while retaining high accuracy. They give preference in later stages to image regions that pass all the classifiers in the previous stages. They also assume that face motion is relatively small from frame to frame. They implemented several additional optimizations: they rescaled image frames rather than features; they used fixed-point arithmetic for coefficients (15 bits) and thresholds (30 bits);

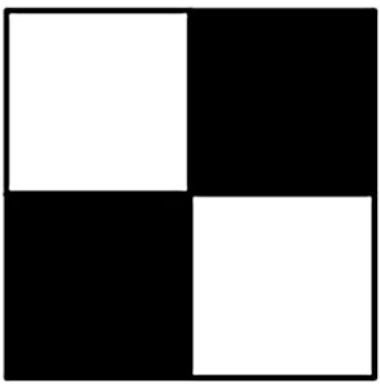


Fig. 40.1 Areas in a $x \times 2$ Haar-like feature

they also used an approximation of the image window normalization factor. Their system architecture includes a classifier pipeline, an image rescaler, RAM for the image windows and for the classifiers, and a control FSM.

40.5.3 Convolutional Neural Networks

A great deal of work has studied the design of Convolutional Neural Network (CNN) algorithms on GPUs. A more recent body of work has developed FPGA-based architectures for CNNs.

Chetlur et al. [6] developed a library of functions for convolutional neural networks implemented on Graphics Processing Units (GPUs). The principal function to be optimized is a spatial convolution which consists of a seven-way nested loops of four independent loops and three accumulation loops. Given that GPUs have a limited amount of onboard memory, their implementation is designed to provide high performance without requiring auxiliary memory. They map the convolutions onto a matrix multiplication formulation to leverage efficient implementations of the matrix multiply. Mapping the convolution components into the matrix requires a nontrivial indexing scheme that is performed as the data is loaded into the GPU memory from off chip. They reported performance on three different convolution algorithms in the range of 1.5-2 TFlops on the Maxwell-based GTX 980.

NVIDIA [16] presents the energy and performance of Tegra X1-based CNNs for inference. They make use of the GPU's FP16 reduced-precision floating-point format to reduce memory bandwidth. Because inference is performed one image at a time, the layer activations are represented by a vector. In contrast, classification is typically performed on batches of images, resulting in an array of layer activations. matrix-vector operations are potentially less efficient than matrix-matrix operations. To minimize inefficiencies, they make use of a generalized matrix-vector product operation. For Alexnet, they reported that their Tegra X1 implementation with 16-bit floating point ran at 258 images/sec compared to 242 images/sec for an Intel Core i7 6700 K; the Tegra X1's efficiency was 45.0 images/sec/W compared to 3.9 images/sec/W for the Core i7 6700 K.

Zhang et al. [35] used loop analysis to optimize the design of an FPGA-based CNN. Their design makes use of the CNN architecture of Krizhevsky et al. [12]. They use a roofline model to guide their exploration of the design space. Given a graph of available performance *vs.* computation to communication ratio, the roof line of the design space is bounded by two limiting curves: at low computation/communication, performance is limited by I/O bandwidth; at high computation/communication, performance is limited by internal computing resources. They use polyhedral data dependency analysis to generate a set of loop structures that vary in their loop scheduling and tile size. They analyzed data reuse to optimize the use of on-chip memory. Their final architecture has a set of processing elements whose inputs and outputs are connected to crossbars. Inputs are fed to the PEs through two sets of input buffers. Similarly, two sets of output buffers capture outputs. Their design provided substantially higher performance density, as

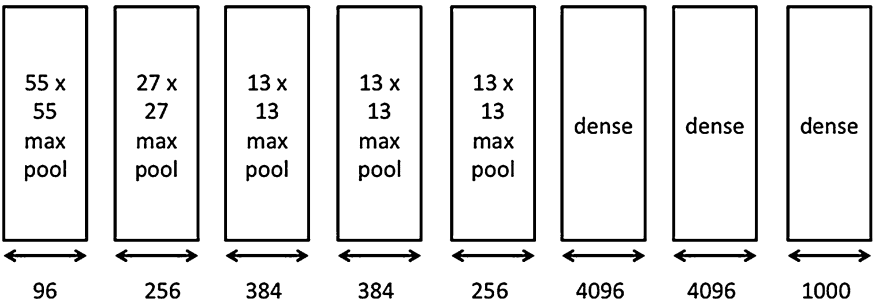


Fig. 40.2 Structure of Microsoft CNN FPGA [17]

measured in Giga Operations Per Second (GOPS) per slice, than previously reported designs. Their design also consumed 18.61 W as compared to 95 W for an Intel Xeon running at 2.2 GHz.

Ovtcharov et al. [17] describe the design of an FPGA-based CNN. Their design is based on the CNN topology of Krizhevsky et al. [12]. As shown in Fig. 40.2, their architecture uses a set of processing elements to perform dot-product operations. The results of the PEs are reordered by a network on chip and returned to the input buffer for the next round of processing. An external DRAM provides bulk storage. Their design is based on the Stratix V D5 FPGA augmented with 8 GB of DDR3 DRAM. Their design was tested on several data sets, including ImageNet 1 K. For that benchmark, their system ran at 134 images/sec at 25 W; in comparison, a Tesla K20 implementation ran at 376 images/sec at 235 W.

40.6 Comparison

We can compare these various approaches to embedded computer vision system design in several aspects.

Programmability of these approaches varies widely. GPUs use nontraditional programming models but do provide a fully programmable model; the characteristics of their memory systems is one important constraint on their practical programmability. Many systems make use of fully hardwired accelerators. A fixed accelerator provides high-performance and low-power consumption. These systems are, however, limited in two ways: they can operate on only limited data sizes and their data and control flow cannot be modified. FPGAs provide high density and flexibility but not true programmability. MPSoCs provide heterogeneous architectures for system designers who are unable to design their own hardware platforms.

Power consumption of these approaches also varies widely. Accelerators typically provide the highest power efficiency. The power efficiency of FPGAs continues to improve as their levels of integration increase. GPU power consumption is higher than that of hardwired systems but considerably lower than that provided by CPUs.

40.7 Design Methodologies

Embedded computer vision systems require translating concepts across a very deep stack of abstractions, from recognition requirements down to software and logic. Design methodologies must keep in mind the large number of translation steps and focus on the right tasks at appropriate points in the design process. Due to the complexity of the design process, embedded CV design tends to be more iterative, combining top-down and bottom-up phases, than are some other types of system design. Ozer and Wolf [18] described the design of one embedded computer vision system.

Embedded computer vision systems typically start with algorithm development in a standard language such as Matlab or OpenCV. The algorithm flow must be determined at this point – what sequence of steps is applied to the image stream.

Architecture planning starts during algorithm development and continues beyond into system design. Some aspects of the algorithms will be guided by the available architectural choices. In many cases, parameter values such as window sizes will be chosen at these early stages. As the algorithm block diagram develops, system designers can start to decide what implementation style is best suited for each block.

Memory system design is a critical component of architecture planning. Computer vision systems require large memory bandwidths. The feasibility of real-time operation may be determined by memory bandwidth; power consumption is also driven by the memory system. Not only must overall bandwidth be satisfied, but the architecture must provide pathways that allow all components to obtain the bandwidth they need. Real-time systems cannot solve memory bottlenecks by sequentializing operations – the required concurrency guides the design of the memory system and internal interconnect.

Hardware and software implementation must pay careful attention to the design goals for each block that were determined during the architecture planning phase. Module design must be tailored to three key design goals: real-time performance, memory performance, and power consumption. Algorithms may be adjusted during this phase as the limits of the platform become clear. A certain number of trade-offs between accuracy, real-time performance, and power can only be made at this later stage when more information is available.

40.8 Conclusion

Heterogeneous architectures are a good match for embedded computer vision given the wide range of algorithms used in the vision pipeline. MPSoCs are widely used, leveraging both programmable cores and video-oriented accelerators. FPGAs have recently gained attention as platforms well-suited to convolutional neural networks.

References

1. Amir A, Zimet L, Sangiovanni-Vincentelli A, Kao S (2005) An embedded system for an eye-detection sensor. Comput Vis Image Underst 98(1):104–123. doi:10.1016/j.cviu.2004.07.009. Special issue on Eye Detection and Tracking
2. Ansaloni G, Bonzini P, Pozzi L (2011) Egra: a coarse grained reconfigurable architectural template. IEEE Trans Very Large Scale Integr VLSI Syst 19(6):1062–1074. doi:10.1109/TVLSI.2010.2044667
3. Bouwens F, Berekovic M, Kanstein A, Gaydadjiev G (2007) Architectural exploration of the adres coarse-grained reconfigurable array. In: Reconfigurable computing: architectures, tools and applications. LNCS, vol 4412. Springer, pp 1–13
4. Casares M, Velipasalar S, Pinto A (2010) Light-weight salient foreground detection for embedded smart cameras. Comput Vis Image Underst 114(11):1223–1237. doi:10.1016/j.cviu.2010.03.023. Special issue on Embedded Vision
5. Chellappa R, Bhattacharyya S, Saha S, Wolf W, Aggarwal G, Schlessman J, Kianzad V (2005) An architectural level design methodology for embedded face detection. In: Third IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES+ISSS'05, pp 136–141. doi:10.1145/1084834.1084872
6. Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E (2014) cuDNN: efficient primitives for deep learning. CoRR abs/1410.0759. http://arxiv.org/abs/1410.0759
7. Clemons J, Jones A, Perricone R, Savarese S, Austin T (2011) Effex: an embedded processor for computer vision based feature extraction. In: 2011 48th ACM/EDAC/IEEE design automation conference (DAC), pp 1020–1025
8. Farabet C, Martini B, Corda B, Akselrod P, Culurciello E, LeCun Y (2011) Neuflow: a runtime reconfigurable dataflow processor for vision. In: 2011 IEEE Computer Society conference on computer vision and pattern recognition workshops (CVPRW), pp 109–116. doi:10.1109/CVPRW.2011.5981829
9. Gudis E, Lu P, Berends D, Kaighn K, van der Wal G, Buchanan G, Chai S, Piacentino M (2013) An embedded vision services framework for heterogeneous accelerators. In: 2013 IEEE conference on computer vision and pattern recognition workshops (CVPRW), pp 598–603. doi:10.1109/CVPRW.2013.90
10. Horprasesert T, Harwood D, Davis LS (1999) A statistical approach for real-time robust background subtraction and shadow detection. In: IEEE international conference on computer vision FRAME-RATE workshop
11. Texas Instruments (2015) TMS320DM816x DaVinci Digital Media Processors Technical Reference Manual, SPRUGX8C, March 2015
12. Krizhevsky A, Sutskever I, Hinton GE (2013) Imagenet classification with deep convolutional neural networks. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ (eds) Advances in neural information processing systems 25. NIPS 2012: neural information processing systems. https://books.google.com/books?id=glsymwEACAAJ

13. Lanuzza M, Perri S, Corsonello P, Margala M (2007) A new reconfigurable coarse-grain architecture for multimedia applications. In: 2007 second NASA/ESA conference on adaptive hardware and systems, AHS 2007, pp 119–126. doi:10.1109/AHS.2007.10
14. Lee MH, Singh H, Lu G, Bagherzadeh N, Kurdahi FJ, Filho EM, Alves VC (2000) Design and implementation of the morphosys reconfigurable computing processor. J VLSI Signal Process Syst Signal Image Video Technol 24(2):147–164
15. Lucas B, Kanade T (1981) An iterative image registration technique with an application to stereo vision. In: International joint conference on artificial intelligence. AAAI
16. nVidia (2015) GPU-based deep learning inference: a performance and power analysis. Technical report
17. Ovtcharov K, Rowase O, Kim JY, Fowers J, Straus K, Chung ES (2015) Accelertaing deep convolutional neural networks using specialized hardware. http://research.microsoft.com/pubs/240715/CNN
18. Ozer B, Wolf M (2014) A train station surveillance system: challenges and solutions. In: 2014 IEEE conference on computer vision and pattern recognition workshops, pp 652–657. doi:10.1109/CVPRW.2014.99
19. Park H, Park Y, Mahlke S (2009) Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In: 42nd annual IEEE/ACM international symposium on microarchitecture, 2009 MICRO-42, pp 370–380
20. Saha S, Puthenpurayil S, Schlessman J, Bhattacharyya S, Wolf W (2008) The signal passing interface and its application to embedded implementation of smart camera applications. Proc IEEE 96(10):1576–1587. doi:10.1109/JPROC.2008.928744
21. Saha S, Bambha NK, Bhattacharyya SS (2010) Design and implementation of embedded computer vision systems based on particle filters. Comput Vis Image Underst 114(11):1203–1214. doi:10.1016/j.cviu.2010.03.018. Special issue on Embedded Vision
22. Schlessman J, Wolf M (2015) Tailoring design for embedded computer vision applications. Computer 48(5):58–62. doi:10.1109/MC.2015.145
23. Soderquist P, Leeser M (1997) Division and square root: choosing the right implementation. Micro IEEE 17(4):56–66. doi:10.1109/40.612224
24. Stein G, Rushinek E, Hayun G, Shashua A (2005) A computer vision system on a chip: a case study from the automotive domain. In: Proceedings of IEEE Computer Society conference on computer vision and pattern recognition – workshops (CVPR 2005), pp 130–130. doi:10.1109/CVPR.2005.387
25. Tabkhi H, Bushey R, Schirner G (2014) Function-level processor (FLP): a high performance, minimal bandwidth, low power architecture for market-oriented MPSoCs. IEEE Embed Syst Lett 6(4):65–68. doi:10.1109/LES.2014.2327114
26. van der Wolf P, Geuzebroek J (2011) SoC infrastructures for predictable system integration. In: Design, automation test in Europe conference exhibition (DATE), 2011, pp 1–6. doi:10.1109/DATE.2011.5763146
27. van der Wolf P, Henriksson T (2008) Video processing requirements on SoC infrastructures. In: Design, automation and test in Europe, 2008, DATE '08, pp 1124–1125. doi:10.1109/DATE.2008.4484827
28. Viola P, Jones M (2001) Rapid object detection using a boosted cascade of simple features. In: Proceedings of the 2001 IEEE Computer Society conference on computer vision and pattern recognition, CVPR 2001, vol 1, pp I–511–I–518. doi:10.1109/CVPR.2001.990517
29. Weber WD, Chou J, Swarbrick I, Wingard D (2005) A quality-of-service mechanism for interconnection networks in system-on-chips. In: Proceedings of the design, automation and test in Europe, vol 2, pp 1232–1237. doi:10.1109/DATE.2005.33
30. Wolf W, Ozer B, Lv T (2002) Smart cameras as embedded systems. IEEE Comput 35(9):48–53
31. Wolf W, Jerraya A, Martin G (2008) Multiprocessor system-on-chip (MPSoC) technology. IEEE Trans Comput Aided Des Integr Circuits Syst 27(10):1701–1713. doi:10.1109/T-CAD.2008.923415

32. Xu J, Wolf W, Henkel J, Chakradhar S, Lv T (2004) A case study in networks-on-chip design for embedded video. In: Proceedings of the design, automation and test in Europe conference and exhibition, vol 2, pp 770–775. doi:10.1109/DATE.2004.1268973
33. Xu J, Wolf W, Henkel J, Chakradhar S (2006) A design methodology for application-specific networks-on-chip. ACM Trans Embed Comput Syst 5(2):263–280. doi:10.1145/1151074.1151076
34. Yang M, Crenshaw J, Augustine B, Mareachen R, Wu Y (2010) Adaboost-based face detection for embedded systems. Comput Vis Image Underst 114(11):1116–1125. doi:10.1016/j.cviu.2010.03.010. Special issue on Embedded Vision
35. Zhang C, Li P, Sun G, Guan Y, Xiao B, Cong J (2015) Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays, FPGA '15, pp 161–170. ACM, New York. doi:10.1145/2684746.2689060

Index

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9