Massimo Torquati · Koen Bertels
Sven Karlsson · François Pacull *Editors*

Smart Multicore Embedded Systems

Springer

Smart Multicore Embedded Systems

Massimo Torquati • Koen Bertels • Sven Karlsson
François Pacull
Editors

Smart Multicore Embedded Systems

Springer

Editors
Massimo Torquati
Computer Science Department
University of Pisa
Pisa, Italy

Sven Karlsson
Informatics and Mathematical Modeling
DTU Informatik
Lyngby, Denmark

Koen Bertels
Delft University of Technology
Delft, The Netherlands

François Pacull
Commissariat à l'énergie atomique et aux énergies alternatives
CEA LetiMinatec
Grenoble, France

ISBN 978-1-4614-8799-9 ISBN 978-1-4614-8800-2 (eBook)
DOI 10.1007/978-1-4614-8800-2
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2013952910

© Springer Science+Business Media New York 2014
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Single-core chip is no longer able to offer increasing performance. Hardware manufactures moved their attention from increasing clock frequency and instruction-level parallelism to thread-level parallelism by designing chips with a number of multiple interconnected cores.

The multi-core technologies are going to rapidly move towards massively parallel computing elements (hundreds or thousands of cores), becoming more and more pervasive in all fields of the embedded system industry, thanks to the improvements of the performance per watt ratio. However, it is clear that increasing the number of cores does not automatically translate into increasing performance.

Old sequential code will get no performance benefit from new multi-core chip. Indeed, since the single-core complexity and the clock frequency are typically lower with respect to old chips, performance of sequential code may get even worse.

The only way to increase performance on multi-core platforms is by squeezing their real power: using thread-level parallelism. However, new multi-threaded parallel code may not scale well with the number of cores due to the overhead introduced for data movement, scheduling and synchronisations. Even parallel code well tuned for maximising performance for a given platform may face the problem of achieving a good trade-off between development costs and time to solution. Parallel software engineering has engaged this challenge mainly by way of designing new tools and tool chains, introducing high-level sequential language extensions and reusable, well-known, parallel patterns.

This book discusses both basic concepts on parallel programming and advanced topics and issues related to the use of real embedded applications. The book derives from the experience and results obtained in the 3-year European ARTEMIS project SMECY (Smart Multi-core Embedded Systems, project number 100230) involving 27 partners in nine European countries. This represents a concrete experience of work in the embedded system area, where application, tool and platform providers worked together in a coordinated way with the goal to obtain new high-level programming tool chains for current and forthcoming embedded many-core platforms. The aim of the book is to describe a "new way" for programming complex applications for embedded many-core systems.

The partners involved in SMECY project were:

France:
Commissariat à l'énergie Atomique et aux 'energies alternatives,
THOMSON Video Networks SA,
Silkan,
STMicroelectronics (Grenoble 2) SAS,
Thales Research & Technology,
Université Joseph Fourier Grenoble 1

The Netherlands:
ACE Associated Compiler Experts bv,
Technische Universiteit Delft

Greece:
Aristotle University of Thessaloniki,
Hellenic Aerospace Industry S.A.,
University of Ioannina

Czech Republic:
Brno University of Technology,
CIP plus s.r.o.,
Ústav Teorie Informace a Automatizace AV ČR, v.v.i.

Denmark:
Danmarks Tekniske Universitet

Sweden:
Free2Move AB,
Hogskolan i Halmstad,
Realtime Embedded AB,
Saab Microwave Systems

Finland:
Nethawk Oyj,
Valtion Teknillinen Tutkimuskeskus (VTT)

Italy:
Politecnico di Milano Dipartimento di Elettronica e Informazione,
Politecnico di Torino,
SELEX E S,
STMicroelectronics S.r.l.,
Alma Mater Studiorum - Universita di Bologna

United Kingdom:
United Kingdom: Thales Research & Technology

SMECY envisioned that multi-core technology will affect and shape the whole business landscape, thus having the ambitious mission to develop and propose new parallel programming technologies enabling the exploitation of embedded many-core architectures. The conceptual approach proposed by SMECY is based on the simple assumption that a tool chain should take both the application requirements and the platform specificities into account in order to be efficient. One of the main outcomes of the project was the definition of an intermediate representation

among front-end and back-end tools defining three different tool chains, one for each application domain.

The book contents are structured as follows:

In Chap. 1 we give a survey on the different parallel programming models available today. These models are suitable not only for general-purpose computing but also for programming specialised embedded systems that offer the required facilities.

Chapter 2 presents the big pictures of the SMECY project with the interaction of the different tools from the applications used as case studies to the targeted hardware platforms. The two-level intermediate representation allowing the interoperability of the components of the tool chains while providing a frame that drastically decreases the amount of work that would be required to add a new tool for further enrichment of the project tool chains is also presented. The intermediate representation designed within the SMECY project represents a bridge between a number of front-end and back-end tools.

Chapters 3 and 4 present in detail the two target platforms considered in the SMECY project, the STHORM platform by STMicroelectronics and the ASVP platform by UTIA.

Chapter 5 covers fault tolerance aspects in the context of embedded systems, whereas Chap. 6 proposes a lightweight code generation process that enables the capability to perform data-dependent optimisations, at runtime, for processing kernels.

Chapters 7, 8 and 9 describe the experiences made developing real-world applications on the SMECY target platform: a radar signal processing application, an object recognition image processing application and a foreground recognition video processing application, respectively.

Our sincere hope is that the reader could find value in the topics covered in this book.

Pisa, Italy — Massimo Torquati
Delft, The Netherlands — Koen Bertels
Lyngby, Denmark — Sven Karlsson
Grenoble, France — François Pacull

Acknowledgements

This book is the fruit of the SMECY project and thus would not have been possible without the involvement of engineers, researchers and technicians who have participated in the project during the last 3 years. In particular, we wish to thank both the work package leaders and the cluster leaders for their coordination effort: Michel Barreteau (WP1), Julien Mottin (WP2), Giovanni Agosta (WP3), Claudia Cantini (WP4 and C1), Athanasios Poulakidas (C2) and Petr Honzik (C3). We would also like to thank the authors who have contributed to the different chapters of this book.

The SMECY project has been partially supported by ARTEMIS Joint Undertaking (project number 100230) and by the national authorities of the nine countries involved: France, the Netherlands, Greece, Czech Republic, Denmark, Sweden, Finland, Italy and United Kingdom.

Massimo Torquati wishes to thank Selex ES and in particular Ing. Filippo De Stefani for the opportunity given to him to work in the great team of people of the SMECY project. He would also like to thank Prof. Marco Vanneschi and Prof. Marco Danelutto from the Computer Science Department of the University of Pisa for their valuable help and support.

Contents

Contributors

Aggelis Aggelis
Hellenic Airspace Industries, Tanagra, Greece

Giovanni Agosta
Politecnico di Milano, Milano, Italy

Michel Barreteau
Thales Research & Technology, Campus Polytechnique - 1, Palaiseau, France

Roman Bartosiński
UTIA AV CR, v.v.i., Praha, Czech Republic

Koen Bertels
Computer Engineering Laboratory, Delft University of Technology, Delft, The Netherlands

Marius Bozga
UJF-Grenoble 1/CNRS, VERIMAG UMR 5104, Grenoble, France

Claudia Cantini
Selex ES, Roma RM, Italy

Mickael Cartron
CEA, LIST, Laboratoire de Fiabilisation des Systèmes Embarqués, Gif-sur-Yvette, France

Henri-Pierre Charles
CEA, LIST, DACLE/LIALP, Grenoble, France

George Chasapis
Faculty of Engineering School of Electrical and Computer Engineering, Aristotle University of Thessaloniki, Thessaloniki, Greece

Damien Couroussé
CEA, LIST, DACLE/LIALP, Grenoble, France

Martin Daněk
UTIA AV CR, v.v.i., Praha, Czech Republic

Vassilios V. Dimakopoulos
Department of Computer Science and Engineering, University of Ioannina, Ioannina, Greece

Petr Honzík
CIP plus, s.r.o., Pribram, Czech Republic

Leoš Kafka
UTIA AV CR, v.v.i., Praha, Czech Republic

Sven Karlsson
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Ronan Keryell
SYLKAN Wild Systems, Los Altos, CA, USA

Lukáš Kohout
UTIA AV CR, v.v.i., Praha, Czech Republic

Victor Lomüller
CEA, LIST, DACLE/LIALP, Grenoble, France

Antonio Miele
Politecnico di Milano, Milano, Italy

Julien Mottin
CEA, LETI, DACLE/LIALP, Grenoble, France

François Pacull
CEA, LETI, DACLE/LIALP, Grenoble, France

Pascal Schleuniger
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Jaroslav Sýkora
UTIA AV CR, v.v.i., Praha, Czech Republic

Massimo Torquati
Computer Science Department, University of Pisa, Pisa, Italy

Giulio Urlini
STMicroelectronics, Milan, Italy

Introduction

Massimo Torquati, Koen Bertels, Sven Karlsson, and François Pacull

1 Embedded Systems

Embedded systems are computational devices whose primary aim is not to serve as traditional general-purpose computers but rather to perform a few dedicated functions. Such devices have become truly ubiquitous and can be found everywhere: in consumer electronics, airplanes, automobiles, trains, toys, security systems, weapon and communications systems and buildings, to just mention a few of their many application areas.

Embedded systems are commonly engaged with the physical world and must often respond to environmental stimuli in real time. The interactions between processes external and internal to the embedded system can be based on the actual time of day, i.e. certain operations are carried out at specific times according to a time schedule. Alternatively, tasks can be performed in response to certain asynchronous events which do not necessarily happen at a specified time. Furthermore, a set of tasks may be carried out within a predetermined time based on interaction with

M. Torquati
Computer Science Department, University of Pisa, Largo B. Pontecorvo, 3 I-56127, Pisa, Italy
e-mail: torquati@di.unipi.it

K. Bertels
Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands
e-mail: k.l.m.bertels@tudelft.nl

S. Karlsson
DTU Compute, Technical University of Denmark, Matematiktorvet, 2800 Lyngby, Denmark
e-mail: svea@dtu.dk

F. Pacull
CEA, LETI, DACLE/LIALP, F-38054, Grenoble, France
e-mail: francois.pacull@cea.fr

the environment. In other words, synchronisation with the environment can be time based, event based or based on human interaction.

Real-time systems place an emphasis on producing a response to an input within a certain time frame. Not meeting the deadline on every occasion is a malfunction in a hard real-time system while it does not necessarily compromise program correctness in a soft real-time system. In the latter case the missed deadlines correspond to degradation in performance or in the quality of service delivered. It is not uncommon for embedded systems to contain a mixture of time-based, event-based and interactive tasks where some have hard real-time constraints while others merely have soft real-time constraints.

Compared to traditional computers, embedded systems are also commonly subject to more tight constraints on physical dimensions, energy consumption, security and reliability. Operations or tasks which have strict deadlines obviously place stronger constraints on the hardware and software and thus become a greater challenge to embedded systems developers. In order to meet constraints and restrictions due to the targeted application domain, embedded processors have highly complex extensions with no counterparts in general-purpose processors.

2 Power as a Driving Force

Power is unarguably the pervasive driving force in today's design of embedded processors. Power requirements are the reason for employing multiprocessor system-on-chip (MPSOC) with specialised hardware for specific tasks such as audio and video processing. Today's MPSOCs, especially in mobile and embedded applications, are shaped by the requirement to do complex calculations at low power. There are many techniques to achieve those requirements and these can be combined to fit the application best.

In video surveillance systems a reliable and uninterrupted power supply is required even in the case of critical situations such as those related to fire, earthquake, burglary, violence and other emergencies. In normal operation, the video system is connected to the power grid and power is not such a big concern. In the case of emergency, however, the system will have to be able to operate completely autonomously and cannot rely on the power grid. It will have to operate on battery power and also store its images locally. How long the battery has to last depends on the operational requirements. For example, it may depend on the expected response time of fire brigades, police or medical assistance. Typically, at least a couple of hours of uninterrupted power is required. On the other hand, the bigger the battery, the more maintenance it requires and the less reliable it is.

For video surveillance, in particular, there are several ways to reduce the power requirement: reduce the strength of the IR lightning, reduce the frame rate and/or resolution of the camera and reduce the accuracy of the detection software. It should be clear that all of these will have their impact on the quality of the processed images and the choice of power reduction methods again depends on

the operational requirements. It should therefore be clear that in this application area, power efficiency is as important and maybe even more as in day-to-day mobile appliances.

A CMOS gate uses little energy in steady state (this may be a problem by itself, but for the sake of simplicity it will be ignored here). More power is used when the gate switches from on to off and vice versa. The energy per second required for gate switching is linear with the switching frequency and with the square of the voltage difference [1]. At first, this seems good news: if we consider a switching action to be "work" (as in a useful processing action), duplicating the frequency duplicates the amount of work done. Since the energy usage is linear with the frequency, it also duplicates. Doubling the amount of energy per second allows to obtain twice the number of work units done per second. However, due to the fact that to run at higher clock speeds, higher voltages are required, the amount of energy required does rise more than linear with the frequency. The exact amount of the increase is not so important in practice, but the effect is significant enough to prefer low clock speeds to high clock speeds.

Following this general rule, a processor should be running at very low frequencies in order to optimise power consumption. The lower the clock speed, the lower the total energy required to perform all work units and the more work units can be done for a given battery charge. The problem is, of course, that the processor would then run very slow and real-time requirements would not be met. This is where parallelism comes in: at a given clock speed, two processors can do twice the number of work units per second. Since each work unit processed in a processor still takes the same amount of energy (as the frequency does not change by adding a second processor), this does not impact how many work units can be done for a given battery charge. This is probably the most important reason why processor engineers, instead of increasing clock frequency and boosting instruction level parallelism for a single core, are adding more and more cores in a single chip in order to increase the performance per watt ratio.

More parallel processors can be added without reducing efficiency and also the resulting additional computational power allows the clock speed to be reduced; hence the voltage can be reduced and efficiency be improved. This is in fact the driving force for parallel processing in embedded systems: use parallelism to reduce the power it takes to compute each work unit. There are however at least two boundaries that put a stop to adding more and more processors while reducing the clock speeds:

1. The chip die size is not unlimited, and the larger the die size, the higher the production costs are.
2. The parallelisation limits of applications.

The first boundary is not the major issue per se. Today's chip production processes are very efficient, allowing for very high-gate density production. It is currently feasible to integrate more than one billion (10^9) transistors on a single silicon chip.

The second boundary instead represents the real actual barrier. These are probably the two most significant problems in reaping the pay-off of parallelisation of the hardware. There are many more, such as the complexity of writing correct parallel applications and the reduced portability of parallel programs. In the general case, these problems are not solved yet.

3 Parallelism Exploitation and Performance Portability

There are many limits to the parallelisation of an application or algorithm. What is in general expected from a parallel program in terms of performance is that the time spent to execute the program in parallel using n processing elements is about $1/n$ of the time spent executing the best sequential program that solves the same problem. Several factors impair the possibility to achieve such desirable behaviour. In particular, the features of the problem to be solved have to be taken into account. Above all, not all the problems are suitable to be parallelised. Some problems have been recognised as inherently sequential and therefore will never benefit from parallel implementation. Furthermore, even those problems suitable to be somehow parallelised may contain a fraction of the work needed to solve the problem, which is non-parallelisable at all.

A theoretical limit is given by the well-known Amdahl's law [2], which states that the amount of non-parallelisable work in an application determines the maximum speedup we may achieve by parallelising that application.[1] If for example 10 % of the application is sequential by nature, even infinite parallelism will not speed up the application by more than a factor of ten.

However, there are many factors that limit the possibility of achieving the theoretical optimum established by Amdahl's law. One main issue is that parallelisation implies the allocation of parts of the program to the different available hardware resources and to manage the communication between the different processing elements. This means that more work needs to be done by the parallel application with respect to its sequential counterpart. This is called the "overhead" introduced by parallel programming. Overhead incurs in the execution of any non-functional code of parallel applications. How much this overhead is depends very much on the way how the application is parallelised and the efficiency by which its parallelism can be mapped to the target architecture. In any case, this overhead will eat into the profits that were made by reducing the clock speed on parallel cores. If we consider again Amdahl's law, it is easy to see that when taking into account overheads, the real limit in the speedup of a parallel application is even smaller than the one determined by the fraction of the non-parallelisable part. It is clear therefore that the

[1] We explicitly do not refer to Gustafson's law which is valid for embarrassingly parallel applications such that the problem size can scale with the number of processors.

lower overhead we introduce in our parallel application with non-functional code, the better performance we can achieve.

The overhead introduced in the parallelisation is strictly related to the tools and programming models used. The scarcity of good high-level programming tools and environments very often forces the application developers to rewrite the sequential programs into parallel software taking into account all the (low-level) features and peculiarities of the underlying platforms. This makes developing efficient parallel applications extremely difficult and time-consuming, and even more important, the resulting code is difficult to maintain and to port with the same or better performance on next-generation architectures. This last aspect is critical because of the rapid evolution of embedded multi-core architectures. In the near future it will be increasingly important that an application is easily portable without the need to re-design from scratch entire parts of the application in order to obtain the same level of performance.

Performance portability is the ability to easily restructure a parallel application, statically and/or dynamically, in order to exploit architectural advances in a cost-effective way. This implies that the parallel application (together with its run-time support) is modified in an automatic way or with a very limited assistance of the application designer. Performance portability is clearly still an open issue. In the best case, the current tools and libraries for parallel programming are only able to guarantee code portability, but they do not typically guarantee performance portability.

From an industrial perspective, the migration of software toward multi-core embedded platforms should be as smooth as possible, and the resulting code must be easy to maintain, independently of the underlying architecture. A useful parallel development environment should thus offer a well-defined programming model and easy-to-use development tools. The programming model should provide a high abstract level allowing to control the execution, communications and synchronisation of concurrent entities, whereas the development tools help developers to write, debug and deploy the parallel applications.

4 The Holy Grail of Parallel Computing

Parallel programs are inherently more difficult to write than sequential ones, because concurrency introduces several new problems that programmers should take into account. Communication and synchronisations among concurrent parts are typically the greatest obstacles to obtain good parallel program performance.

Switching from sequential to modestly parallel computing (4–12 cores) made programming much more difficult due to the new difficulties and overheads introduced, eventually without a comparable payback. Many-core chip multiprocessors (CMP) offer hundreds of much simpler cores and a progress to thousands of cores over time with a more favourable performance per watt ratio.

The main problem is that most algorithms and software implementations are built fundamentally on the premise that concurrency would continue to increase modestly [3]. This makes more and more urgent a change in the current programming practice by software engineers. On the other hand, the difficulties of programming many cores are even more than programming multi-core and for this reason have the parallel programming research community embraced the new challenge of devising new programming models and models of computation for many-core systems.

For years the research community tried to find the holy grail in targeting a unique chain of tools and a single programming model which would work efficiently for any platform, any type of parallelism and any application domain. The underlying idea is to device a set of techniques or tools which are able to both minimise overheads and to guarantee performance portability on next-generation platforms.

The vision proposed in this book is different to a certain extent and is the basis of the European project SMECY[2] which can be best summarised as follows: a "one fits all" tool chain does not exist. A unique compilation tool chain would not be able to target efficiently different many-core platforms for several application domains. The conceptual approach proposed by SMECY is based on the simple statement that a tool chain should take into account both the application requirements and the platform specificities of various embedded systems in different industries in order to achieve efficiency. As a consequence, a suitable parallel programming tool chain has to take as input parameters both the application and the platform models and then it should be able to determine all the relevant transformations and mapping decisions on the concrete platform minimising user intervention and hiding the difficulties related to the correct and efficient use of memory hierarchy and low-level code generation.

4.1 The SMECY Project Vision

The SMECY project has been structured in three clusters focusing research activities and integration effort for two many-core platform and a specific application domain. The platforms considered are:

- The STHORM platform (also known as P2012) developed by STMicroelectronics is a high-performance accelerator. It is organised around multiple clusters implemented with independent power and clock domains, connected via a high-performance fully asynchronous network-on-chip (NoC) which provides scalable bandwidth, power efficiency and robust communication across different power and clock domains. Each cluster features up to 16 tightly coupled processors sharing multi-banked level-1 data memories.

[2]Smart Multi-core Embedded Systems, ARTEMIS project number 100230.

- The ASVP platform is designed by UTIA (the Institute of Information Theory and Automation, Czech Republic) to parallelise and speed up common DSP algorithms based on vector or matrix operations. It is based on a master-worker architecture, where workers are very simple CPUs playing the role of programmable finite-state machines to control configurable data paths, to eliminate execution stalls and to enhance the computation efficiency.

The application domains covered by the SMECY project are:

- Radar signal processing
- Multimedia and mobile video and wireless
- Stream processing applications

A number of front-end and back-end tools are currently available for many embedded systems. In order to obtain a good balance between performance and portability, an intermediate representation (IR) between front-end and back-end tools of the tool chains has been introduced. The IR allows to increase code portability enabling also intermediate code inspection and editing in order to simplify both debugging and low-level code optimisations.

The IR has been designed having in mind the following general properties:

1. It has to be sufficiently generic to allow expressiveness and be compatible with various programming models or model of computation.
2. Its semantics should be sufficiently close to the front-end tools ones and should be compatible with the different target platforms.
3. It should strongly rely on already existing standards in order to be well accepted and to maximise the time life.

In the SMECY project, a two-level IR has been defined; both levels are based on already existing standards. The high-level IR called SME-C is based on standard C with pragmas similar to OpenMP 3.1 [1]; the low-level IR is based on APIs meant to be used by back-end tools for targeting platform code generation based on standard proposed by the Multicore Association.[3] Moreover, some tools are able to ensure the equivalence in between the two levels. Indeed, source-to-source compilers can transform the SME-C source files to pure C with API calls as specified for the low-level IR. The choice to have a two-level IR is mainly due to the different levels of abstraction required by the various tools that span from high-level programming tools to low-level APIs.

[3] http://www.multicore-association.org/.

5 Chapter Summary

The trends within embedded software development can change rapidly in response to the release of new programming tools and frameworks or the introduction of new and more capable computing platforms. The fundamental trade-offs between different programming models and models of computation, however, are less likely to change overnight and thus this book will focused on these trade-offs. Since it is becoming increasingly common that an embedded system contains a number of heterogeneous and homogeneous processing elements, special attention will be paid on the following points by discussing them throughout the book in detail:

- A complete reasoned, state-of-the-art of programming model and model of computation with particular emphasis to the models targeting embedded systems
- An in-depth description of two embedded platforms taken as case study: STMicroelectronis' STHORM platform (also known as P2012) and UTIA's ASVP platform (also known as EdkDSP)
- The definition and the issues of an intermediate representation in the context of embedded systems in order to obtain a good trade-off between performance and portability
- Run-time tools and fault-tolerance techniques
- Three significant application case studies in three distinct application domains: signal processing, image processing and video processing

References

1. J. Rabaey, *Digital Integrated Circuits*. Prentice Hall, 1996.
2. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18–20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.
3. J. Shalf, "The new landscape of parallel computer architecture," *Journal of Physics: Conference Series*, vol. 78, no. 1, pp. 012 066+, 2007.

Part I
Parallel Programming Models and Methodologies

Chapter 1
Parallel Programming Models

Vassilios V. Dimakopoulos

1.1 Introduction

Programming models represent an abstraction of the capabilities of the hardware to the programmer. A programming model is a bridge between the actual machine organization and the software that is going to be executed by it. As an example, the abstraction of the memory subsystem leads to two fundamentally different types of parallel programming models, those based on shared memory and those based on message passing. Programming through shared memory can be compared to the use of a bulletin board. Information is exchanged by posting data to shared locations which are agreed upon a priori by the sender and receiver of data. In shared-memory architectures, this is realized directly using ordinary loads and stores. On the other hand, communication via message passing occurs through point-to-point transfers between two computing entities and is thus conceptually similar to the exchange of letters which explicitly identify the sender and receiver of the information. In distributed-memory architectures, these transfers are realized over an actual processor interconnection network.

Because a programming model is an abstraction of the underlying system architecture, it is usually not tied to one specific machine, but rather to all machines with architectures that adhere to the abstraction. In addition one particular system may support more than one programming model, albeit often not with the same efficiency.

A successful parallel programming model must carefully balance opacity and visibility of the underlying architecture; intricate details and idiosyncrasies should be hidden while features which enable the full computational power

V.V. Dimakopoulos (✉)
Department of Computer Science and Engineering, University of Ioannina,
Ioannina, GR-45110, Greece
e-mail: dimako@cs.uoi.gr

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_1,
© Springer Science+Business Media New York 2014

of the underlying device need to be exposed. The programming model also determines how program parts executed in parallel communicate and which types of synchronization among them are available.

In summary, a programming model enables the developer to express ideas in a certain way. To put the ideas into practice, however, the developer has to choose a specific programming language. The correspondence between programming models and programming languages is close but not one-to-one. Several languages may implement the same programming model. For instance, Java and C++ with Pthreads offer very similar shared-memory programming models although the two languages differ significantly in several other important aspects. Furthermore, it is possible for a programming language and set of APIs to adhere to several programming models.

In what follows we present the most significant parallel programming models available today. In Sect. 1.2 we classify them according to the memory abstraction they offer to the programmer. In Sect. 1.3 we survey models based on shared memory while Sect. 1.4 covers distributed-memory models. Section 1.5 surveys the available models for GPUs and accelerators which represent devices with private memory spaces. Models that try to combine some of the above categories are examined in Sect. 1.6. Finally, Sect. 1.7 visits other promising languages and programming styles that do not fall in the above categories.

1.2 Classification of Parallel Programming Models

Algorithms and whole applications contain parallelism with varying degrees of regularity and granularity. Parallelism can be exploited at the following granularity levels: bits, instructions, data, and tasks. *Bit-level parallelism* is exploited transparently to the programmer by increasing the processor word size thus reducing the number of instructions required to perform elementary arithmetic and bit manipulation operations. Similarly, *instruction-level parallelism* can be exploited by processors and compilers with little or no involvement from the programmer. Processors support overlap of instruction execution through techniques such as instruction pipelining, super-scalar, and out-of-order execution while optimizing compilers can increase the efficiency of these techniques by careful reorderings of program instructions.

At higher granularity levels, such as *data level*, parallelism is not generally exploitable without programmer intervention and the programming model plays a major role. `For` loops in programming languages are a major source of data-level parallelism and therefore this kind of parallelism is also referred to as *loop-level parallelism*. Many processors include single-instruction, multiple-data (SIMD) instructions which operate on short vectors typically containing two to eight data values. These differ from single-instruction, single-data (SISD) instructions as the latter can only compute a single result at a time. Some programming models

allow the programmer to utilize SIMD instructions either explicitly or implicitly, by means of a vectorizing compiler. In any case, it is required that the data-parallel computation can be expressed as operations on vectors, i.e. that a single instruction can be applied to multiple data values.

Another important way of exploiting data-level parallelism is *multiprocessing* or *multi-threading*, where the data-parallel computation is distributed over multiple processors, a single processor capable of executing multiple instruction streams in parallel, or any combination of these. Such architectures are characterized as multiple-instruction, multiple-data (MIMD) machines as they are capable of processing multiple independent instruction streams, each of which operates on separate data items. If data parallelism is found in a loop nest, the workload is distributed among the processing elements by assigning a fraction of the overall iteration count to each of them. Unlike SIMD instructions, the distribution of work can be dynamic if the time to process each of the iterations varies.

The coarsest-grained and least regular kind of parallelism is *task-level parallelism*, which almost uniformly relies on the programmer's ability to identify the parts of an application that may be executed independently. Task-level parallelism is about the distribution of both computation and data among processing elements, whereas data parallelism primarily emphasizes the distribution of data among processing elements. Programming models vary significantly in their support of exploiting task-level parallelism—some are strongly focused on task-level parallelism and provide abstractions which are flexible enough to also exploit data parallelism while others are mainly focused on the exploitation of structured or loop-level parallelism and provide only weak support for irregular and dynamic parallelism at the granularity of tasks.

Except the above considerations on the level and type of parallelism offered, parallel programming models can also be classified by the memory abstraction they present to the programmer. The memory organization of typical parallel computing systems gives rise to two broad architectural categories. In *centralized* or *shared-memory multiprocessors*, the memory is accessible to all processors with uniform access time (UMA). The processors are usually attached to a bus, sharing its bandwidth, an architecture also known as a *symmetric multiprocessor* (SMP). Unfortunately, buses and centralized memories suffer from increased contention and longer latencies when more processors are added. This limited scalability means that central memories are only used in smaller multiprocessors, usually having up to eight processors. They are also found in the individual nodes of larger systems.

In the second organization, the memory is physically distributed among processors. Two access policies are then possible; each local memory either is accessible only to the respective processor or can be accessed by all processors albeit with nonuniform memory access (NUMA) times through a global address space. The former is called a (pure) *distributed-memory organization*, whereas the latter is a *distributed shared-memory organization* which can be cache coherent (ccNUMA) or not. While scalable ccNUMA systems are a challenge to design, they put smaller burden on the programmer because replication and coherence is managed transparently

by the hardware. Non-cache-coherent distributed shared-memory systems require either the programmer to make sure that each processor view of the memory is consistent with main memory or the intervention of a software layer which manages the consistency transparently in return for some decrease in performance.

The programming models used to target a given platform are closely related to the underlying memory and communication architecture. Shared-memory models offer the programmer the ability to define objects (e.g. variables) that are accessible to all execution entities (e.g. processes or threads) and can be realized most efficiently on machines containing hardware which keeps the memories and caches coherent across the system. Explicit message passing, the dominant representative of distributed-memory models, is commonly used whenever memory is non-centralized and noncoherent. However, both shared- and distributed-memory organizations can support any kind of programming model, albeit with varying degrees of efficiency. For instance, it is possible to support message passing very efficiently by passing a pointer rather than copying data, on shared-memory architectures. A global shared-memory model can also be supported on a distributed-memory machine either in hardware (ccNUMA) or in software with a runtime layer [2], although obtaining acceptable speedups on a wide range of unmodified, parallel applications is challenging and still a subject of active research.

The parallel-computing landscape has been augmented with systems that usually operate as back-end attachments to a general-purpose machine, offering increased execution speeds for special portions of code offloaded to them. *General-purpose graphical processing units* (GPGPUs) and *accelerators* are typical examples. Because of their distinct memory and processing element organization these devices warrant suitable programming models. In addition, their presence gives rise to *heterogeneous* systems and programming models in the sense that the host and the back-end processing elements are no longer of the same type and the programmer must provide different programs for each part of the system.

It is also possible to combine the above paradigms and thus obtain a *hybrid programming model*. This usually requires the programmer to explicitly utilize multiple and different programming styles in the same program and is, for example, popular when targeting clusters of multicores/multiprocessors. Another possibility is represented by the Partitioned Global Address Space (PGAS) languages. Here the notions of shared and distributed memory are unified, treating memory as a globally shared entity which is however logically partitioned, with each portion being local to a processor.

Finally, many other different attributes can be considered in order to classify parallel programming models [3]. For example, they can be categorized based on their application domain or by the way execution entities (workers) are defined, managed, or mapped to actual hardware. Another attribute is the programming paradigm (procedural, object-oriented, functional, streaming, etc). In this chapter we focus on the memory abstraction offered by the programming model.

1.3 Shared-Memory Models

Shared-memory models are based on the assumption that the execution entities (or workers) that carry the actual execution of the program instructions have access to a common memory area, where they can store objects, uniformly accessible to all. They fit naturally UMA architectures (SMPs/multicores with small processor/core counts) and ccNUMA systems. Shared-memory models can be further categorized by the type of execution entities employed and the way they are handled by the programmer (explicit or implicit creation, mapping and workload partitioning).

1.3.1 POSIX Threads

A *thread* is an autonomous execution entity, with its own program counter and execution stack which is usually described as a "lightweight" process. A normal ("heavyweight") process may generate multiple threads; while autonomous, the threads share the process code, its global variables, and any heap-allocated data. The Portable Operating System Interface (POSIX) provides for a standard interface to create threads and control them in a user program. POSIX is a standardization of the interface between the operating system and applications in the Unix family of operating systems. The POSIX.1c thread extensions [4] provide a description of the syntax and semantics of the functions and data types used to create and manipulate threads and is known as *Pthreads* [5].

In parallel applications, multiple threads are created by the `pthread_create` call. Because the execution speed and sequence of different threads is unpredictable and unordered by default, programmers must be aware of race conditions and deadlocks. Synchronization should be used if operations must occur in certain order. POSIX provides *condition variables* as their main synchronization mechanism. Condition variables provide a structured means for a thread to wait (block) until another thread signals that a certain condition becomes true (`pthread_condition_wait`/`pthread_condition_signal` calls). The real-time extensions to POSIX [6] define *barriers* as an additional synchronization mechanism for Pthreads. A thread that calls `pthread_barrier_wait` blocks until all sibling threads perform the same call; they are all then released to continue their execution.

Pthreads provide *mutex objects* as a primary way of achieving mutual exclusion, which is typically used to coordinate access to a resource which is shared among multiple threads. A thread should lock (`pthread_mutex_lock`) the mutex before entering the critical section of the code and unlock it (`pthread_mutex_unlock`) right after leaving it. Pthreads also provide *semaphores* as another mechanism for mutual exclusion.

Pthreads are considered a rather low-level programming model that puts too much burden on the programmer. Explicitly managing and manipulating the execution entities can sometimes give ultimate control to an expert programmer [5], but this comes at the cost of increased complexity and lower productivity since larger Pthreads programs are considered hard to develop, debug, and maintain.

1.3.2 OpenMP

Because threads are a versatile albeit a low-level primitive for parallel programming, it has been argued that they should not be used directly by the application programmer; one should rather use higher-level abstractions which are possibly mapped to threads by an underlying runtime. OpenMP [1] can be seen as a realization of that philosophy. It is a set of compiler directives and library functions which are used to parallelize sequential C/C++ or Fortran code. An important feature of OpenMP is its support for *incremental* parallelism, whereby directives can be added gradually starting from a sequential program.

Like Pthreads, OpenMP is an explicitly parallel programming model meaning that the compiler does not analyze the source code to identify parallelism. The programmer instructs the compiler on how the code should be parallelized, but unlike Pthreads, in OpenMP threads are not an explicit notion; the programmer primarily creates and controls them implicitly through higher-level directives.

OpenMP supports a fork-join model of parallelism. Programs begin executing on a single, master thread which spawns additional threads as parallelized regions are encountered (enclosed in an `omp parallel` directive), i.e. a fork. Parallel regions can be nested although the compiler is not required to exploit more than one level of parallelism. At the end of the outermost parallel region the master thread joins with all worker threads before continuing execution.

A parallel region essentially replicates a job across a set of spawned threads. The threads may cooperate by performing different parts of a job through *worksharing* constructs. The most prominent of such constructs is the `omp for` (C/C++) or `omp loop` (Fortran) directive where the iterations of the adjacent loop are divided and distributed among the participating threads. This allows for easy parallelization of regular loop nests and has been the main strength of OpenMP since these loops are prevalent in scientific codes such as linear algebra or simulation of physical phenomena on rectangular grids.

Since revision 3.0 of the standard [7] the applicability of OpenMP has been significantly broadened in applications with dynamic and irregular parallelism (e.g. when work is created recursively or is contained in loops with unknown iteration counts) with the addition of the `omp task` directive. Tasks are blocks of code that are marked by the programmer and can be executed asynchronously by any thread. Because of their asynchronous nature, tasks must also carry a copy of the data they will operate on when actually executed.

It would not be an exaggeration to say that OpenMP has nowadays become the de facto standard for shared-memory programming. It is used to program

multiprocessors and multicores alike, whether they physically share memory or they have ccNUMA organizations. It has also been implemented successfully for a number of embedded platforms (e.g. [8–10]). There even exist implementations of earlier versions of OpenMP which target computational clusters though shared virtual memory software libraries (albeit without reaching high performance levels). Another important fact is that the directive-based programming style of OpenMP and its latest addition of tasks have a profound influence on many recent programming model proposals for others architectures (cf. Sect. 1.5). However, notice that while an intuitive model to use, OpenMP may not always be able to produce the maximum performance possible since it does not allow fine low-level control.

1.4 Distributed-Memory Models

Distributed-memory systems with no physical shared memory can be programmed in a multitude of ways. To name a few:

- Low-level socket programming
- Remote procedure calls (e.g., SUN RPC, Java RMI)
- Software shared virtual memory [2], to provide the illusion of shared memory
- Message passing

are among the models that have been used. However, *message passing* is by far the dominating programming model for developing high-performance parallel applications in distributed architectures.

Notice also that approaches similar to the above have also been proposed in specific domains. For example, in the context of real-time software for embedded heterogeneous MPSoCs, such as multimedia and signal processing applications, the TTL [11] and Multiflex [12] frameworks provide programming models based on tasks or objects communicating by transferring tokens over channels or through remote procedure calls.

1.4.1 Message-Passing (MPI)

The message-passing model assumes a collection of execution entities (processes, in particular) which do not share anything and are able to communicate with each other by exchanging explicit messages. This is a natural model for distributed-memory systems where communication cannot be achieved through shared variables. It is also an efficient model for NUMA systems where, even if they support a shared address space, the memory is physically distributed among the processors.

Message passing is now almost synonymous to MPI, the Message Passing Interface [13, 14]. MPI is a specification for message-passing operations and is implemented as a library which can be used by C and Fortran programs. An MPI program consists of a number of identical processes with different address spaces,

where data is partitioned and distributed among them (single-program, multiple-data or SPMD style). Interaction among them occurs through messaging operations. MPI provides send/receive operations between a named sender and a named receiver, called point-to-point communications (`MPI_Send` / `MPI_Recv`). These operations are cooperative or *two-sided*, as they involve both sending and receiving processes, and are available in both *synchronous* and *asynchronous* versions.

A synchronous pair of send/receive operations defines a synchronization point between the two entities and requires no buffering since the sender remains blocked until the transfer completes. If the synchronous send/receive pair is not executed simultaneously, either the sender or receiver blocks and is prevented from performing useful work. Asynchronous message passing allows the sender to overlap communication with computation thus increasing performance if the application contains enough exploitable parallelism. In this case buffers are required, and depending on timing, caution is needed to avoid filling them up.

The second version of the MPI [14] added a number of enhancements. One of the most significant is the ability to perform one-sided communications (`MPI_Put`/`MPI_Get`), where a process can perform remote memory accesses (writes or reads) without requiring the involvement of the remote process.

There also exists a very rich collection of global or *collective* (one-to-many, many-to-many) operations such as *gather*, *scatter*, *reduction*, and *broadcast* which involve more than two processes and are indispensable for both source code structuring and performance enhancement.

MPI dominates programming on computational clusters. Additionally, there exist implementations that allow applications to run on larger computational grids. There also exist lightweight implementations specialized for embedded systems, such as LMPI [15]. MPI is generally considered an efficient but low-level programming model. Like Pthreads, the programmer must partition the work to be done by each execution entity and derive the mapping to the actual processors. Unlike Pthreads, one also needs to partition and distribute the data on which the program operates.

1.5 Heterogeneity: GPGPU and Accelerator Models

General-purpose graphics processing units (GPGPUs) employ the power of a GPU pipeline to perform general-purpose computations instead of solely graphical operations. They have been recognized as indispensable devices for accelerating particular types of high-throughput computational tasks exhibiting data parallelism. They consist typically of hundreds to thousands of elementary processing cores able to perform massive vector operations over their wide vector SIMD architecture.

Such devices are generally nonautonomous. They assume the existence of a *host* (CPU) which will *off-load* portions of code (called *kernels*) for them to execute. As such, a user program is actually divided in two parts—the host and the device

part, to be executed by the CPU and the GPGPU correspondingly, giving rise to heterogeneous programming. Despite the heterogeneity, the coding is generally done in the same programming language which is usually an extension of C.

The two dominant models for programming GPGPUs are Compute Unified Device Architecture (CUDA) and OpenCL. The first is a programming model developed by NVIDIA for programming its own GPUs. The second is an open standard that strives to offer platform-independent general-purpose computation over graphics processing units. As such it has also been implemented on non-GPU devices like general or special-purpose accelerators, the ST P2012/STHORM being a characteristic example [16]. Other models have also been proposed, trying to alleviate the inherent programming heterogeneity to some degree.

1.5.1 CUDA

In CUDA [17, 18] the computation of tasks is done in the GPU by a set of threads that run in parallel. The threads are organized in a two-level hierarchy, namely, the *block* and the *grid*. The first is a set of tightly coupled threads where each thread is identified by a thread ID while the second is a set of loosely coupled blocks with similar size and dimension. The grid is handled by the GPU, which is organized as a collection of "multiprocessors." Each multiprocessor executes one or more of the blocks and there is no synchronization among the blocks.

CUDA is implemented as an extension to the C language. Tasks to be executed on the GPU (kernels) are functions marked with the new `__global__` qualifier. Thread management is implicit; programmers need only specify grid and block dimensions for a particular kernel and do not need to manage thread creation and destruction. On the other hand, workload partitioning and thread mapping is done explicitly when calling the `__global__` kernel using the `«<gs,bs»>` construct, where `gs` (`bs`) specifies the dimensions of the grid (block).

The memory model of CUDA consists of a hierarchy of memories. In particular, there is per-thread memory (registers and *local* memory), per-block memory (*shared* memory, accessed by all threads in a block), and per-device memory (read/write *global* and read-only *constant* memory, accessed by all threads in a grid and by the host). Careful data placement is crucial for application performance.

1.5.2 OpenCL

OpenCL [19] is a standardized, cross-platform, parallel-computing API based on the C99 language and designed to enable the development of portable parallel applications for systems with heterogeneous computing devices. It is quite similar to CUDA although it can be somewhat more complex as it strives for platform independence and portability.

As in CUDA, an OpenCL program consists of two parts: kernels that execute on one or more devices and a host program that manages the execution of kernels. Kernels are marked with the `__kernel` qualifier. Their code is run by *work items* (cf. CUDA threads). Work items form *work groups*, which correspond to CUDA thread blocks. The memory hierarchy is again similar to CUDA with the exception that memory shared among the items of a work group is termed *local memory* (instead of shared), while per-work item memory is called *private* (instead of local).

In OpenCL, devices are managed through *contexts*. The programmer first creates a context that contains the devices to use, through calls like `clCreateContext`. To submit work for execution by a device, the host program must first create a command queue for the device (`clCreateCommandQueue`). After that, the host code can perform a sequence of OpenCL API calls to insert a kernel along with its execution parameters into the command queue. When the device becomes available, it removes and executes the kernel at the head of the queue.

1.5.3 Directive-Based Models

Because programming GPGPUs with the CUDA and OpenCL models is tedious and in a rather low abstraction level, there have been a number of proposals for an alternative way of programming these devices. The common denominator of these proposals is the task-parallel model; kernels are simply denoted as tasks in the user programs and thus blend naturally with the rest of the code, reducing thus the impact of heterogeneity. Moreover, these models are heavily influenced by the OpenMP directive-based style of programming.

GPUSs [20] uses two directives. The first one allows the programmer to annotate a kernel as a task, while also specifying the required parameters and their size and directionality. The second one maps the task to the device that will execute it (in the absence of this directive, the task is executed by the host CPU). This also determines the data movements between the host and the device memories.

Another example is HMPP [21] which offers four types of directives. The first one defines the kernel (termed *codelet*) to be executed on the accelerator. The second one specifies how to use a codelet at a given point in the program, including which device will execute it. The third type of directives determines the actual data transfers before executing the codelet. Finally, a fourth directive is used for synchronization.

Finally, OpenACC [22] is another attempt to provide a simplified programming model for heterogeneous CPU/GPU systems. It is developed by Cray, CAPS, NVIDIA, and PGI and consists of a number of compiler directives and runtime functions. OpenACC is expected to be endorsed by the upcoming version of OpenMP.

1.6 Hybrid Models

Hybrid programming models try to combine two or more of the aforementioned models in the same user program. This can be advantageous for performance reasons when targeting systems which do not fall clearly in one architectural category. A characteristic example is a cluster of multicore nodes. Clusters are distributed-memory machines. Their nodes are autonomous computers, each with its own processors and memory; nodes are connected through an interconnection network which is used for communicating with each other. Within a node, however, the processors (or cores) have access to the same local memory, forming a small shared-memory subsystem.

Using one programming model (e.g. OpenMP combined with software shared virtual memory layers, or MPI only) to leverage such platforms is a valid option but may not be the most efficient one. Hybrid programming utilizes multiple programming models in an effort to produce the best possible code for each part of the system.

A similar situation occurs in systems consisting of multiple nodes of CPUs and/or GPU cards. There have been works that combine, for example, OpenMP and CUDA [23] or even CUDA and MPI [24]. However, this case is not considered in detail here as the various GPU models are more or less hybrid by nature in the sense of already supporting heterogeneous CPU/GPU programming.

1.6.1 Pthreads + MPI

Under this model, Pthreads are used for shared-memory programming within a node while MPI is used for message passing across the entire system. While the first version of the MPI standard was not designed to be safely mixed with user-created threads, MPI-2 [14] allows users to write multithreaded programs easily. There are four levels of thread safety supported by systems and selectable by the programmer (through the `MPI_Init_thread` call): `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. Except for `SINGLE` all other levels allow multiple threads. In the `FUNNELED` level, however, only one of the threads is allowed to place MPI calls. The other two levels allow MPI calls by all threads (albeit nonsimultaneously in the `SERIALIZED` level). Notice that not all implementations support all thread safety levels.

While there have been some application studies reported with the hybrid Pthreads + MPI model (e.g. [25, 26]), this is not an extensively used approach; OpenMP is the most popular choice for spawning shared-memory parallelism within a node.

1.6.2 OpenMP + MPI

This is the most widely used combination, where OpenMP is employed to leverage the shared-memory features of a node [27–29]. This, of course, guarantees portability since both OpenMP and MPI are industry standards. Many application classes can benefit from the usage of this model. For example, some applications expose two levels of parallelism, coarse-grained (suitable for MPI) and fine-grained (suitable for OpenMP). The combination may also help in situations where the load of the MPI processes is relatively unbalanced. Finally, OpenMP threads may provide a faster communication mechanism than multiple MPI processes within a node.

On the other hand, some applications possess only one level of parallelism. In addition, the utilization of OpenMP is not without its costs; it introduces the overheads of thread creation, synchronization, and worksharing. Consequently, the hybrid OpenMP + MPI model may not always be the better choice (see, e.g. [30, 31]).

The user program is structured as a collection of MPI processes. The code of each process is enhanced with OpenMP directives to take advantage of the presence of shared memory. Depending on the programming and the capabilities of the MPI implementation MPI calls may be made by the master thread only, outside `parallel` regions. The other option is to allow MPI calls within `parallel` regions and thus have some thread(s) communicate while others compute (the `SERIALIZED` or `MULTIPLE` safety levels are required). As a result, one can overlap communication with computation. This requires the most complicated control, but can result in performance improvements.

The hybrid OpenMP + MPI model has been used successfully in many cases (e.g. [32–34]). We should also note the existence of frameworks that facilitate programming with this model while also combining it with others, such as task-centric ones (e.g. HOMPI by [35]).

1.6.3 PGAS

PGAS stands for Partitioned Global Address Space and represents a class of programming languages and runtime libraries that try to marry the shared- and distributed-memory models when targeting clusters of SMPs or multicores. However, while the other hybrid approaches force the programmer to mix two different models in the same code, PGAS presents a single, unified model which inherits characteristics of both.

In the PGAS model multiple SPMD threads (or processes) share a part of their address space. However, this globally shared area is logically partitioned, with each portion being local to a processor. Thus, programs are able to use a shared address space, which generally simplifies programming, while also exposing data/computation locality in order to enhance performance. Because of this, PGAS is sometimes termed *locality-aware* shared-memory programming.

Two characteristic examples of the PGAS model are Co-array Fortran (CAF, [36]) and Unified Parallel C (UPC, [37]). The first one is an extension to Fortran 95 and is now incorporated in the recent Fortran 2008 standard. A program in CAF consists of SPMD processes (*images*). The language provides for defining private and shared data, accessing shared data with one-sided communications and synchronization among images. Similarly, UPC extends the C programming language to allow declaring globally shared variables, partitioning and distributing their storage across the available nodes. It offers an affinity-aware loop construct (`upc_forall`) as well as built-in calls for collective communications among threads.

1.7 Other Parallel Programming Models

In this section we briefly discuss other approaches to parallel programming. We take a different view from the previous sections in that we do not categorize them according to their memory abstraction. We first take a look at new languages or notable parallel extensions to well-known sequential languages, irrespectively of the memory model they follow. Then we visit a different way of parallel programming—skeleton-based models.

1.7.1 Languages and Language Extensions

Cilk [38] is a language extension to C which adds support for parallel programming based on tasks or *Cilk procedures*. Syntactically, a Cilk procedure is a regular C function where the `cilk` keyword has been added to its definition to allow asynchronous invocation. Cilk procedures can then be invoked or spawned by prefixing the function invocation with the `spawn` keyword. A key strength of Cilk is its support for irregular and dynamic parallelism. Work stealing, a provably optimal technique for balancing the workload across processing elements, was developed as part of the Cilk project. It has subsequently been adopted by numerous other parallel programming frameworks. Cilk++ is a commercial implementation of the language.

Sequoia [39] is a language extension to a subset of C. In Sequoia tasks are represented via a special language construct and are isolated from each other in that each task has its own address space and calling a subtask or returning from a task is the only means of communication among processing elements. To achieve isolation tasks have call-by-value-result semantics and the use of pointer and reference types inside tasks is disallowed. First-class language constructs are used to express data decomposition and distribution among processing elements and locality-enhancing transformations such as loop blocking; for instance, the `blkset` data type is used to represent the tiles or blocks of a conventional array. Another characteristic of Sequoia is that the memory hierarchy is represented explicitly via trees of memory

modules. Generic algorithmic expression and machine-specific optimizations are kept (mostly) separate. The source code contains tunable parameters and variants of the same task which are optimized for different hardware architectures. A separate set of mapping files stores values of tunable parameters and choices of task variants for the individual execution platform.

Hierarchically tiled arrays (HTA, [40]) are an attempt to realize efficient parallel computation at a higher level of abstraction solely by adding specialized data types in traditional, imperative languages such as C++. As the name implies, HTAs are hierarchies of tiles where each tile is either a conventional array or an HTA itself. Tiles serve the dual purpose of controlling data distribution and communication at the highest level and of attaining locality for the sequential computation inside each task. Like Sequoia, HTA preserves the abstraction of a global shared memory while automatically generating calls to a message-passing library on distributed-memory architectures.

Java is one of the most popular application-oriented languages. However, it also provides support for parallel programming. For one, it was designed to be multi-threaded. Additionally, Java provides Remote Method Invocation (RMI) for transparent communication among virtual machines. A number of other programming models can also be exercised using this language. For example, *MPJ Express* [41] is a library that provides message-passing facilities to Java. Finally, Java forms the basis for notable PGAS languages such as *Titanium* [42].

1.7.2 Skeletal Programming

Algorithmic skeletons [43] represent a different, higher-level pragmatic approach to parallel programming. They promote *structured parallel programming* where a parallel program is conceived as two separate and complementary concerns: *computation*, which expresses the actual calculations, and *coordination*, which abstracts the interaction and communication. In principle, the two concepts should be orthogonal and generic, so that a coordination style can be applied to any parallel program, coarse- or fine-grained. Nonetheless, in conventional parallel applications, computation and coordination are not necessarily separated, and communications and synchronization primitives are typically interwoven with calculations.

Algorithmic skeletons essentially abstract commonly used patterns of parallel computation, communication, and interaction (e.g. map-reduce, for-all, divide and conquer) and make them available to the programmer as high-level programming constructs. Skeletal parallel programs can then be expressed by interweaving *parametrized* skeletons using composition and control inheritance throughout the program structure. Based on their functionality, skeletons can be categorized as data parallel (which work on bulk data structures and typically require a function or sub-skeleton to be applied to all elements of the structure, e.g. *map* or *reduce*), task-parallel (which operate on tasks, e.g. *farm*, *pipe*, *for*), or resolution (which outline algorithmic methods for a family of problems, e.g. *divide and conquer* or

branch and bound). Notice that the interface of the skeleton is decoupled from its implementation and as a consequence the programmer need only specify *what* is to computed and not *how* it should be deployed on a given architecture.

Most skeleton frameworks target distributed-memory platforms, e.g. eSkel [44], SKELib [45], and ASSIST [46], which deliver task- and data-parallel skeletal APIs. Java-based Skandium [47] and C++-based FastFlow [48], on the other hand, target shared-memory systems. The FastFlow framework is differentiated as it focuses on stream parallelism, providing farm, divide and conquer, and pipeline skeletons. A detailed survey of algorithmic skeleton frameworks is given in [49].

1.8 Conclusion

In this chapter we attempted to outline the parallel programming models landscape by discussing the most important and popular ones. The reader should be aware that what we presented is just a portion of what is available for the programmer or what has been proposed by researchers. We classified the models mainly according to the memory abstraction they present to the programmer and then presented the representative ones in each category. We covered shared-memory models, distributed-memory models, and models for GPUs and accelerators. Hybrid models combine the aforementioned ones in some way. We finally concluded with a summary of other models that do not fit directly in the above categories.

Parallel programming models is a vast area of active research. The multitude of platforms and architectures and the variety of their combinations are overwhelming but at the same time a source of new problems and ideas. The domination of multi- and many-core systems even on the desktop has pushed research on parallel programming even further. This is not without reason since the "concurrency revolution is primarily a software revolution. The difficult problem is not building multicore hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in CPU performance" [50].

References

1. OpenMP ARB, "OpenMP Application Program Interface V3.1," July 2011.
2. J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *IEEE Concurrency*, vol. 4, pp. 63–79, 1996.
3. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.
4. IEEE, "IEEE P1003.1c/D10: Draft standard for information technology – Portable Operating System Interface (POSIX)," Sept 1994.

5. D. Butenhof, *Programming With Posix Threads*, ser. Addison-Wesley Professional Computing Series. Addison-Wesley, 1997.
6. IEEE, "IEEE Std 1003.1j-2000: Standard for information technology – Portable Operating System Interface (POSIX)- part 1: System Application Program Interface (API)-Amendment J: Advanced real-time extensions," pp. 1–88, 2000.
7. OpenMP ARB, "OpenMP Application Program Interface V3.0," May 2008.
8. F. Liu and V. Chaudhary, "A practical OpenMP compiler for system on chips," in *WOMPAT '03, Int'l Workshop on OpenMP Applications and Tools*, Toronto, Canada, 2003, pp. 54–68.
9. T. Hanawa, M. Sato, J. Lee, T. Imada, H. Kimura, and T. Boku, "Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP," in *IWOMP 2009, 5th International Workshop on OpenMP*, Dresden, Germany, June 2009, pp. 15–27.
10. S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying OpenMP on an embedded multicore accelerator," in *SAMOS XIII, 13th Int'l Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation*, Samos, Greece, July 2013.
11. P. Vanăder Wolf, E. deăKock, T. Henriksson, W. Kruijtzer, and G. Essink, "Design and programming of embedded multiprocessors: an interface-centric approach," in *Proc. CODES+ISSS '04, 2nd IEEE/ACM/IFIP Int'l Conference on Hardware/software Codesign and System Synthesis*, New York, USA, 2004, pp. 206–217.
12. P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu, "Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management," in *Proc. CODES+ISSS '04, 2nd IEEE/ACM/IFIP Int'l Conference on Hardware/software Codesign and System Synthesis*, New York, USA, 2004, pp. 48–53.
13. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. Cambridge, MA: MIT Press, 1999.
14. W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA: MIT Press, 1999.
15. A. Agbaria, D.-I. Kang, and K. Singh, "LMPI: MPI for heterogeneous embedded distributed systems," in *ICPADS '06, 12th International Conference on Parallel and Distributed Systems - Volume 1*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 79–86.
16. L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012, pp. 983–987.
17. NVIDIA, *NVIDIA CUDA Programming Guide 2.0*, 2008.
18. D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.
19. Khronos Group, *The OpenCL Specification Version 1.0*, Beaverton, OR, 2009.
20. E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the StarSs programming model for platforms with multiple GPUs," in *Euro-Par '09, 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 851–862.
21. R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," in *GPGPU 2007, Workshop on General Purpose Processing on Graphics Processing Units*, 2007.
22. OpenACC, *The OpenACC™ Application Programming Interface Version 1.0*, Nov 2011.
23. Y. Wang, Z. Feng, H. Guo, C. He, and Y. Yang, "Scene recognition acceleration using CUDA and OpenMP," in *ICISE '09, 1st International Conference on Information Science and Engineering*, 2009, pp. 1422–1425.
24. Q.-k. Chen and J.-k. Zhang, "A stream processor cluster architecture model with the hybrid technology of MPI and CUDA," in *ICISE, 2009, 1st International Conference on Information Science and Engineering*, 2009, pp. 86–89.
25. C. Wright, "Hybrid programming fun: Making bzip2 parallel with MPICH2 & Pthreads on the Cray XD1," in *CUG '06, 48th Cray User Group Conference*, 2006.

26. W. Pfeiffer and A. Stamatakis, "Hybrid MPI / Pthreads parallelization of the RAxML phylogenetics code," in *9th IEEE International Workshop on High Performance Computational Biology*, Atlanta, GA, Apr 2010.
27. L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *Scientific Programming*, vol. 9, no. 2,3, pp. 83–98, Aug. 2001.
28. R. Rabenseifner, "Hybrid parallel programming on HPC platforms," in *EWOMP '03, 5th European Workshop on OpenMP*, Aachen, Germany, Sept 2003, pp. 185–194.
29. B. Estrade, "Hybrid programming with MPI and OpenMP," in *High Performance Computing Workshop*, 2009.
30. F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks," in *SC '00, ACM/IEEE Conference on Supercomputing*, Dallas, Texas, USA, 2000.
31. D. S. Henty, "Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling," in *SC '00, ACM/IEEE Conference on Supercomputing*, Dallas, Texas, USA, 2000.
32. K. Nakajima, "Parallel iterative solvers for finite-element methods using an OpenMP/MPI hybrid programming model on the earth simulator," *Parallel Computing*, vol. 31, no. 10–12, pp. 1048–1065, Oct. 2005.
33. R. Aversa, B. Di Martino, M. Rak, S. Venticinque, and U. Villano, "Performance prediction through simulation of a hybrid MPI/OpenMP application," *Parallel Comput.*, vol. 31, no. 10–12, pp. 1013–1033, Oct. 2005.
34. P. D. Mininni, D. Rosenberg, R. Reddy, and A. Pouquet, "A hybrid MPI-OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence," *Parallel Computing*, vol. 37, no. 6–7, pp. 316–326, 2011.
35. V. V. Dimakopoulos and P. E. Hadjidoukas, "HOMPI: A hybrid programming framework for expressing and deploying task-based parallelism," in *Euro-Par'11, 17th International Conference on Parallel processing*, Bordeaux, France, Aug 2011, pp. 14–26.
36. R. W. Numrich and J. Reid, "Co-arrays in the next Fortran standard," *SIGPLAN Fortran Forum*, vol. 24, no. 2, pp. 4–17, Aug. 2005.
37. UPC Consortium, "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.
38. M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI '98, ACM SIGPLAN 1998 conference on Programming language design and implementation*, Montreal, Quebec, Canada, 1998, pp. 212–223.
39. K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06, 2006 ACM/IEEE Conference on Supercomputing*, Tampa, Florida, 2006.
40. G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *PPoPP '06, 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, New York, USA, 2006, pp. 48–57.
41. A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core HPC systems using Java," *J. Parallel Distrib. Comput.*, vol. 69, no. 6, pp. 532–545, Jun. 2009.
42. K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11–13, pp. 825–836, 1998.
43. M. Cole, *Algorithmic skeletons: structured management of parallel computation*. London: Pitman / MIT Press, 1989.
44. M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.
45. M. Danelutto and M. Stigliani, "SKElib: parallel programming with skeletons in C," in *Proc. of 6th Intl. Euro-Par 2000 Parallel Processing*, ser. LNCS, A. Bode, T. Ludwing, W. Karl, and R. Wismüller, Eds., vol. 1900. Munich, Germany: Springer, Aug. 2000, pp. 1175–1184.
46. M. Vanneschi, "The programming model of ASSIST, an environment for parallel and distributed portable applications," *Parallel Computing*, vol. 28, no. 12, pp. 1709–1732, 2002.

47. M. Leyton and J. M. Piquer, “Skandium: Multi-core programming with algorithmic skeletons,” in *PDP ’10, 18th Euromicro Int’l Conference on Parallel, Distributed and Network-Based Processing*, 2010, pp. 289–296.
48. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, “FastFlow: high-level and efficient streaming on multi-core,” in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds. Wiley, Jan. 2013, ch. 13.
49. H. González-Vélez and M. Leyton, “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers,” *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.
50. H. Sutter and J. Larus, “Software and the concurrency revolution,” *ACM Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.

Chapter 2
Compilation Tool Chains and Intermediate Representations

Julien Mottin, François Pacull, Ronan Keryell, and Pascal Schleuniger

2.1 Introduction

In SMECY, we believe that an efficient tool chain could only be defined when the type of parallelism required by an application domain and the hardware architecture is fixed. Furthermore, we believe that once a set of tools is available, it is possible with reasonable effort to change hardware architectures or change the type of parallelism exploited.

2.1.1 Application Domains

In the SMECY consortium, the application providers have selected more than 15 applications. These have been clustered into the following three sets:

Radar Signal Processing

- Passive coherent location, PCL, radar
- Space-time adaptive processing, STAP

J. Mottin (✉) • F. Pacull
CEA, LETI, DACLE/LIALP, F-38054, Grenoble, France
e-mail: julien.mottin@cea.fr; francois.pacull@cea.fr

R. Keryell
SYLKAN Wild Systems 4962 El Camino Real 201 Los Altos, CA 94022, USA
e-mail: Ronan.Keryell@silkan.com

P. Schleuniger
DTU Compute, Technical University of Denmark, Matematiktorvet, 2800 Lyngby, Denmark
e-mail: pass@dtu.dk

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_2,
© Springer Science+Business Media New York 2014

- Active electronically scanned array, AESA, radar
- Measurement and analysis of time-varying targets

Radar applications reconstruct a view of the world from echoes of radio signals. In the SMECY project, both active and passive radar approaches have been studied. The structure of the radar applications varies but generally consists of an input filtering stage followed by a stage which correlates echoes to real-world features. The correlation phase tends to be very computational intensive but also requires high memory performance.

Multimedia, Mobile, and Wireless Transmission

- Orthogonal frequency division multiplexing, OFDM, modem
- Audio decoding
- Software spectrum sniffer for cognitive radio systems in the ISM Band.
- Mobile network protocol analyzer
- Video processing on mobile nodes

The applications in this set are more diverse than the applications in the radar set. Three applications address radio communication. These three range from data coding applications to a spread spectrum radio sniffer. The mobile aspect is represented with an application which performs distributed video processing on mobile devices. The final application in the set is an analyzer for network protocols with high demands on throughput.

Stream Processing (Video Surveillance)

- Video surveillance
- Video encoding
- Object detection
- Wavelet image processing
- High dynamic range image processing
- 3D graphics particle rendering
- Time-space features processing

The final set of applications focus on high performance image and video processing. All the applications have strict requirements on latency and are commonly used in a data-streaming fashion. While the algorithms used in the applications are diverse, they are all computational intense due to high image resolutions and strict latency requirements.

2.1.2 Target Platforms

For each application set, we have chosen, as target, one of the two platforms provided by the project partners: P2012, also known as STHORM, provided

by STMicroelectronics and EdkDSP, and its last version ASVP, provided by the academic partner UTIA, Institute of Information Theory and Automation in Czech Republic. These two targets are very different and so require different tool chains. The main difference between the platforms is that P2012 is based on replicated patterns interconnected by a NoC, network on chip, while EdkDSP utilizes heterogeneous hardware accelerators.

The two first sets of applications used P2012 and the third EdkDSP.

2.2 The Tool Chains

One of the major outcomes of the SMECY project is the interaction of the different tools provided by the partners as assets at the beginning of the project and the ones developed on purpose during the project. The former have been adapted to use the defined intermediate representation.

Figure 2.1 illustrates the big picture with the three application sets: green for radar signal processing on P2012, red for multimedia, mobile, and wireless transmission also on P2102, and finally blue for stream processing on EdkDSP. We call the application sets, *clusters*.

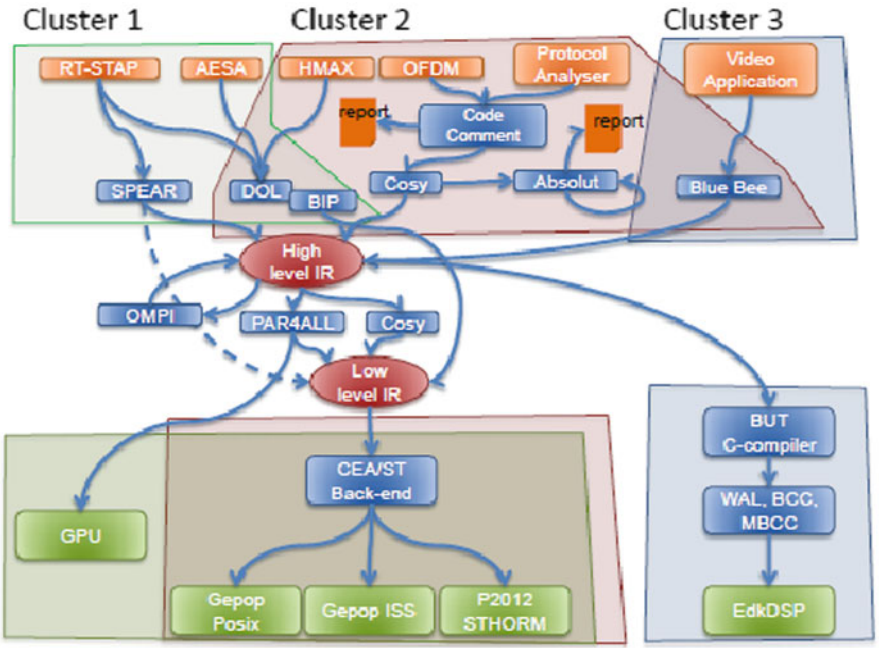

Fig. 2.1 The three tool chains

On top, in orange, we can find the applications from the three application sets previously are introduced. On the bottom, in light green, we have the two target platforms: EdkDSP and P2012, plus some other platform used as reference and some simulators for the P2012 platforms. In the middle, in blue, we have all the tools we considered in the project:

- Front-end tools that guide the programmer in his/her work of mapping the application on a multicore system. This part is independent of a given physical architecture and considers only an abstraction of the actual hardware.
- Back-end tools, that are dedicated to a given hardware in order to obtain the best performances out of a given multicore platform.

In between the front-end and back-end tools, in red, is a key result of the project: the two level intermediate representations presented into more details hereafter.

2.3 Intermediate Representations

The SMECY intermediate representations have been developed to be sufficiently generic to allow expressiveness and be compatible with various programming models or model of computation. We decided on two intermediate representations, both capitalizing on existing standards:

- A high-level intermediate representation, called *SME-C*, based on standard C with pragmas similar to OpenMP.
- A low-level intermediate representation, called *IR2*, meant to be used by back-end tools for target platform code generation. The representation is based on standard C with a set of APIs based on a standard proposed by the Multicore Association.

We decided on this multitiered approach because a single intermediate representation is not a good match for the different levels of abstraction used by the tools in the tool chains. Furthermore, we have developed tools that can transform between the two intermediate representations. Examples include Par4All, by SILKAN-project, and CoSy, by ACE, which can transform SME-C source files to the low-level intermediate representation.

We will now describe the two intermediate representations.

2.3.1 High Level Intermediate Representation: SME-C

The SME-C representation is a #pragma-based language extending C and C++ to allow heterogeneous computing with several accelerators and memory spaces. SME-C is based on OpenMP to express parallelism on a main host, extended with some specific #pragma to invoke accelerated functions on a given accelerator or

to pipeline loops. The programming model is based on C processes, with a virtual shared memory and threads *à la* OpenMP. SME-C includes mapping information stating on which hardware part a function is to be placed and run.

An important part of SME-C is the support for describing memory dependencies at the function call level. Memory dependencies are approximated with rectangles, more generally hyperparallelepipede in any dimension, in multidimensional arrays. The hyperparallelepipede is tagged as read, written, or both. With this information, the tools can infer the memory communication. This approach is similar to XMP pragmas and high performance Fortran, HPF.

SME-C leverages the TR 18037 Embedded C standard to express detailed hardware properties such as hardware register names and precision of fixed-point arithmetic computations.

In several of the SMECY applications, data is processed in a pipelined fashion however with strong data dependencies that require data to be processed sequentially at specific points in the pipeline. This is opposed to a data-parallel model, where such dependencies do not exist. A streaming model is a suitable parallel computation model for such programming patterns.

Streaming models can exploit both a coarse-grained level of parallelism and parallelism at a fine-grained level. At the fine-grained level, the overhead of passing data between processing nodes in the pipeline must of course be minimized. For fine-grained parallelism, this may require hardware support. To achieve good load balancing, it is important that the processing nodes have a comparable grain size. If one node required much more processing than the others, it becomes the bottleneck and no parallelism can be exploited.

Streaming has the advantage of data locality. Data is passed around in the distributed point-to-point network. Such networks can be implemented far more efficiently than, for example, a shared memory connection between the processing nodes.

SME-C supports streaming through annotations. The communication is derived from the program source and generated by the compiler. This is very important for the parallel performance tuning of the application because it allows for experimentation with different load partitioning without having to reprogram process communication.

2.3.2 Low-Level Intermediate Representation: IR2

The low-level intermediate representation, called *IR2*, is based on an API developed by the Multi-core Association [1,2]. The Multi-core Association has developed APIs for communication, resource management, and task management. In the SMECY project, we have used the communication API called *MCAPI*. This improves interoperability and allows us to leverage the effort the Multi-core Association has put into developing the API. The low-level intermediate representation consists of C source code with MCAPI calls. MCAPI is based on three main concepts:

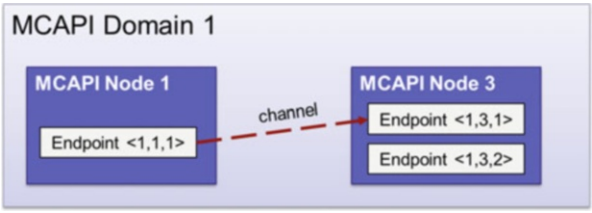

Fig. 2.2 MCAPI main concepts

- A *node* is an independent thread of control that can communicate with other nodes. The exact nature of a node is defined by the implementation of the MCAPI. It could, for example, be a process, a core, a thread, or an HW IP.
- An *endpoint* is a communication termination point and is therefore connected to a node. One node can have multiple endpoints, but one endpoint belongs to a single node.
- *Domain* is a set of MCAPI nodes that are grouped together for identification or routing purpose. The semantics attached to a domain is given by the implementation. Each node can only belong to a single domain.

Compared to MPI, MCAPI offers inter-core communication with low latency with minimal footprint. MCAPI communication nodes are all statically predefined by the implementation.

MCAPI offers three fundamental communication mechanisms:

- *Messages* are streams sent from one endpoint to another. No connection is established between the two endpoints to send a message. This is the easiest way of communicating between two nodes.
- A *packet channel* is a FIFO unidirectional stream of data packets of variable size, sent from one endpoint to another.
- A *scalar channel* is similar to a packet channel, except that only fixed-length word of data can be sent through the channel. A word may be 8, 16, 32, or 64 bits of data.

Figure 2.2 presents an overview of MCAPI. The communication channels can then be set up between two different endpoints.

Within the project, we have developed an implementation of MCAPI for the STHORM platforms. The main objective of this implementation was to offer a uniform level of abstraction and a homogeneous programming interface for the whole platform.

2.3.3 Source-to-Source Compilers

In the project, we have developed two source-to-source compilers whose role is to translate from the high-level intermediate representation to the low-level intermediate representation. The two compilers are complementary and translate

different parts of SME-C. The first one is dedicated to the streaming annotation outlined above. The second source-to-source compiler developed translates the rest of SME-C.

In addition, a more general OpenMP source-to-source compiler has been developed.

Source-to-Source for Streaming Annotation (ACE)

ACE has implemented a source-to-source compiler that accepts C programs with the streaming annotation and produces a partitioned program with separate processes, nodes, for each stage of the streaming pipeline. In addition, it implements the communication links between the nodes.

The generated code includes library calls to implement the low-level tasks of process creation and synchronization. This small library of about five calls is currently implemented on top of, shared memory, POSIX pthreads. It is not hard to retarget this library to different underlying runtime systems.

To be used, the streaming model requires a part of the target application to be rewritten into a particular form, using a while loop and the SME-C stream annotations. Only this part needs to be passed through the source-to-source compiler. Hence, large parts of the application remain unmodified and do not need to be processed by the stream compiler.

Stream termination is currently not handled well. Stream termination has to be mapped from a sequential to a distributed decision process and the design and implementation of that is still to be done.

It is possible to stream a `while` loop in several pipeline stages that execute in parallel and pass information between stages.

The two pragmas used here are:

`#pragma smecy stream_loop`: this indicated the following `while` loop must be turned into a stream of processes.

`#pragma smecy stage`: this acts as a separator between groups of statements and defines the boundary of pipeline stages. Only data passing over these separators is turned into communication.

Smecc Source-to-Source (SILKAN)

Smecc is a source-to-source translator from SME-C pragma-oriented C and C++ to OpenMP and IR2. It is based on the ROSE compiler. The translator is intended to be used as a front-end to the Par4All compiler.

Par4All is an automatic parallelizing and optimizing compiler developed notably by SILKAN (HPC Project). Par4All uses a set of macros and API functions collectively called *Par4All Accel runtime* to ease parallelized code generation by masking implementation-dependent or hardware-dependent details. Par4All can parallelize to several CPUs, using OpenMP, or GPUs, using Cuda and OpenCL.

OMPi Source-to-Source (UOI)

OMPi is a lightweight source-to-source OpenMP compiler and runtime system for C, conforming to version 3.0 of the specifications. The OMPi compiler takes C source code annotated with OpenMP #pragmas and produces transformed multithreaded C code, ready to be compiled by the native compiler of the system. It provides a multitude of runtime libraries for supporting efficient execution. OMPi supports shared memory systems, threads, or processes, with loop or task-based parallelism. Task-based parallelism can be combined with message passing for a hybrid model. OMPi yields a performance improvement through OpenMP parallelization and sophisticated dynamic runtime scheduling.

2.4 The Tools

We conclude with brief descriptions of each tool in the tool chains.

2.4.1 Front-End Tools

BIP (VERIMAG)

BIP (Behavior, Interaction, Priority) is a formal component-based framework, which allows building complex systems by coordinating the behavior of a set of atomic components [3]. Atomic components are described as Petri-nets extended with data and functions described in C. The BIP toolbox includes translators from various programming models into BIP, source-to-source transformers for BIP, as well as a configurable compiler for generating code executable by a dedicated middle-ware engine. The BIP framework and toolbox aims to support the design flow for embedded applications. The tool accepts input in BIP language and outputs debugging functionalities, deadlock analysis, performance analysis based on simulation, and C/C++ implementation for general purpose platforms, according to different options (real-time, single/multithreaded, monolithic, distributed, ...).

ABSOLUT (VTT)

ABSOLUT is a tool to perform system-level performance exploration. It considers standard multithreaded C code as input programming language. ABSOLUT creates an abstract workload model and simulates the workload model on performance

capacity model of the target platform. The approach enables early performance evaluation, exhibits light modeling effort, allows fast exploration iteration, and reuses application and platform models. It also provides performance results that are accurate enough for system-level exploration.

BlueBee (TUDelft)

BlueBee is a tool chain that allows (embedded systems) developers to port their applications to heterogeneous multicore platforms. It consists of a portioning and mapping toolbox that determines on the basis of performance driven profiling information what parts of the application should be mapped on what particular computing element. Once the kernels identified, the necessary code transformations are performed to insert the necessary instructions to start and stop the different kernels and to transfer the parameters to the different computing elements. The appropriate back-end tools compile the identified kernels for the different HW units to which they are mapped. The HW units can be FPGAs, DSPs, and GPPs. The partitioning toolbox and the back-end tools can be used independently. BlueBee aims to improve performance through (semi)automatic partitioning and mapping on a heterogeneous multicore platform. BlueBee supports ANSI C as input language and outputs ANSI C with pragmas or binary.

Code Comment (DTU)

DTU has explored the use of tools which provide code comments to the programmer. The tools help the programmers make better use of compilers optimization features. For many parallel applications, performance relies not on instruction-level parallelism, but on loop-level parallelism. Unfortunately, many modern applications are written in ways that obstruct automatic loop parallelization. The aforementioned generated comments guide the programmer in iteratively modifying application source code to exploit the compiler's ability to generate loop-parallel code.

SpearDE (Thales)

SpearDE is a code generation environment for custom parallel embedded architectures. The graphical model-based environment provides the user with both domain-specific application interfaces and a heterogeneous architecture description interface, which help the implementation of data-streaming applications for parallel architectures. Using SpearDE, the programmer should be able to fill the gap between functional and implementation levels by using a seamless design flow that includes modeling of both the application and the target architecture. If needed,

the programmer can try several mapping strategies by using the design space exploration facility provided by Spear. SpearDE provides support to integrate code generators for specific targets (e.g., Thales SIMD architecture on FPGA for image processing), or it can be used to properly interface external tools from a more domain-relevant point of view.

Ptolemy II and HAMMER (HH)

Ptolemy II is a software system for modeling and simulation of concurrent real-time embedded systems. It is developed at University of California, Berkley. The tool has a thematic focus on system design through assembly of concurrent components, which is relying on the use of well-defined models of computation to define the interaction between the components. One of the main research areas on the Ptolemy system concerns the use of heterogeneous mixture of models of computation. However, Ptolemy II provides no direct support for multi- and many-core modeling, scheduling, and code deployment. Ptolemy II's code generator infrastructure got modified and extended during the SMECY project by HH. The system is able to generate input to the back-end tools in the form of the common SMECY IR (intermediate representation).

HAMMER is a multi- and many core analysis and mapping tool. HAMMER is built on top of the Ptolemy II system. It uses the Ptolemy II infrastructure for programming and modeling concurrent functional data flow behavior of applications. HAMMER adds the functionality needed for mapping applications on multi- and many-core systems and for analyzing nonfunctional behavior such as timing analysis.

2.4.2 Back-End Tools

UTIA ASVP SDK (UTIA)

UTIA ASVP SDK consists of a C-compiler, assembler, and linker and two application interfaces to the ASVP accelerators. The first part of the SDK contains a C-compiler and assembler (binutils) for hardware accelerator's microcontroller in the used ASVP platform. The tools compile input C codes to microcontroller firmware binaries. Two APIs represented by header files and libraries are the next part of the SDK. The first API is called WAL (worker abstraction layer) and it defines an interface between the host CPU and the accelerators. It offers functions for data transfer and execution control between the host CPU and local accelerator's data memories. The second API defines an interface between the microcontroller and data flow unit and between the accelerator and the host CPU from the side of the accelerator. It offers functions for communication with the host CPU and functions to parameterize and control basic operations in hardware. The UTIA ASVP SDK

provides target-independent interface between ASVP platform and the partner's tools, namely (1) C-compiler and linker—compilation of a control C code to binary firmware for the accelerator microcontroller and (2) APIs—provide basic interface functions.

BUT C-Compiler (BUT)

The Vecta is a C language compiler that generates code for architectures with accelerator and was developed primarily for the EdkDSP platform. It is a fully automatic compilation chain. This compilation chain can be used as back-end compiler for code prepared by BlueBee and Par4All. The approach to compilation is based on a view that the workers may have a predefined set of possible vector operations that they can perform. This set of operations is based on the WAL API provided by UTIA, where a complex operation may consist of a sequence of provided basic operations defined by this API. Vecta allows using multiple BCEs automatically and also legacy code without much modification can be quickly ported to the EdkDSP. Description of BCEs for the BCE acceleration pass is parameterizable and new operations can be simply added or removed. Also, the operation tree obtained from the for-loop body can be analyzed in order to generate application-specific dataflow units for BCEs. The overall design is prepared to be modified for other architectures with external accelerators. The Vecta C-compiler provides automatic off-loading of expensive for-loops to basic computing elements.

deGoal (CEA)

deGoal is a tool designed to build specialized code generators (also known as compilettes) customized for each computing kernel we want to accelerate in an application. Such compilettes are designed with the aim to perform data- and architecture-dependent code optimizations and code generation at runtime. Furthermore, compilettes provide very fast code generation and low memory footprint. This approach is fundamentally different from the standard approach for dynamic compilation as used, for example, in Java Virtual Machines or LLVM JIT. In order to target computing architectures that include domain-specific accelerators and to raise the level of abstraction of the source code of compilettes, deGoal uses a dedicated programming language, which is later transformed to C language by an automatic source-to-source translation in order to ease integration with standard compilation tool chains.

CoSy Compiler Development System (ACE)

The CoSy compiler development system is a modular toolbox for compiler construction. CoSy is a product that is used by many companies worldwide.

CoSy supports C, C++, and extensions such as Embedded C. The CoSy front-end also accepts OpenMP and can be programmed to accept arbitrary pragma extensions to implement domain-specific programming models. For shared memory multiprocessing, CoSy supports OpenMP's relaxed memory consistency model. CoSy is typically used to generate compilers that produce assembly code from C. CoSy can also generate compilers that manipulate or transform source code, and produce source code again. CoSy aims to create best-in-class compilers in the embedded application domain. For this reason, it supports a wide variety of extensions that are not found in other compilers or compiler systems.

References

1. The Multicore Association, "Industry standards to solve multicore challenges," http://www.multicore-association.org/, 2011.
2. The Multicore Association, "Multicore Communication APIs," http://www.multicore-association.org/workgroup/mcapi.php, 2011.
3. A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 3–12. [Online]. Available: http://dx.doi.org/10.1109/SEFM.2006.27

Part II
HW/SW Architectures Concepts

Chapter 3
The STHORM Platform

Julien Mottin, Mickael Cartron, and Giulio Urlini

3.1 Introduction

The STHORM platform is a high-performance accelerator developed by STMicroelectronics (in the following simply ST) and CEA, which was initially designed under the P2012 project name. This hardware platform comes along with different simulators that allows to test parallel programs on regular computers. Both the hardware platform and the simulators are described in the following sections.

3.2 Platform

The STHORM platform is organized around multiple clusters implemented with independent power and clock domains. They are connected via a high-performance fully asynchronous NoC (network on chip) which provides scalable bandwidth, power efficiency, and robust communication across different power and clock domains. Each cluster features up to 16 tightly coupled processors sharing

J. Mottin (✉)
CEA, LETI, DACLE/LIALP, F-38054, Grenoble, France
e-mail: julien.mottin@cea.fr

M. Cartron
CEA, LIST, Laboratoire de Fiabilisation des Systèmes Embarqués, Point Courrier 94, Gif-sur-Yvette, F-91191, France
e-mail: mickael.cartron@cea.fr

G. Urlini
STMicroelectronics, Milan, Italy
e-mail: giulio.urlini@st.com

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_3,
© Springer Science+Business Media New York 2014

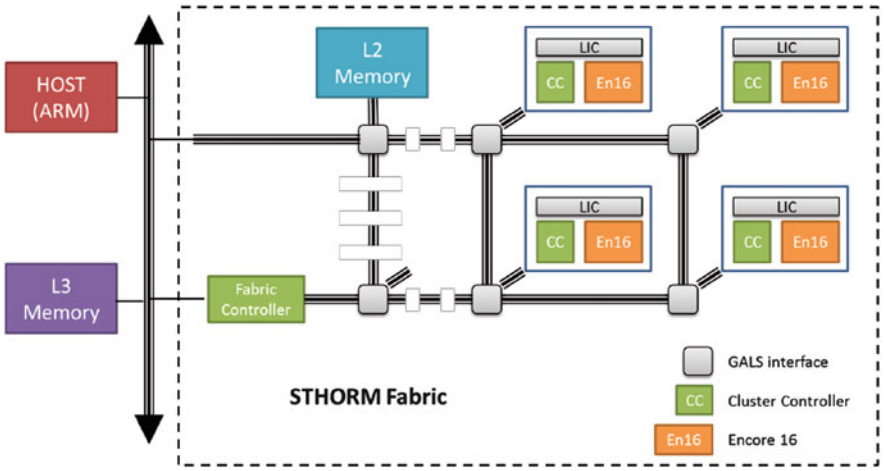

Fig. 3.1 STHORM architecture template

multi-banked level-1 data memories. The current ST roadmap is to deliver an implementation with four clusters: thus the global number of programmable processing elements will be 64. More details can be found in [1].

Figure 3.1 presents the overview of the STHORM platform. It is split in two regions: the fabric region and the host region. The fabric is the actual many-core system and is composed of four clusters, one fabric controller which acts as a main controller for the whole fabric and some fabric shared memory which will be referred as L2 memory. The fabric region is represented inside the dash line in Fig. 3.1.

The host is a dual-core ARM-based platform featuring some peripherals (not shown in the figure) that can communicate with the fabric. The fabric provides a multicore accelerator and the host is a regular dual-core processor that runs a classical operating system (i.e., android) and may host classical libraries. Some DRAM external memory is also available outside the fabric and will be referred as the L3 memory. The L3 memory can provide reasonable space (typically 512 MB or more) while the fabric shared L2 memory is 1 MB.

Figure 3.2 provides more details on the inner cluster organization. It is composed of two main blocks: the Encore<N> ("Enough of Core N"), a block made of N processing elements (PEs) and a 256 KB shared memory, and the cluster controller (CC). The Encore<N> block also contains a hardware synchronizer (HWS) that can be used to implement HW synchronizations such as mutexes and semaphores. In the platform we used, N is equal to 16 which means that 16 PEs per cluster are available for application programming. The CC is used as a central controller for the whole cluster. It can use the DMA block for advanced memory transfers between the different memories. It can also communicate with other clusters and with the host.

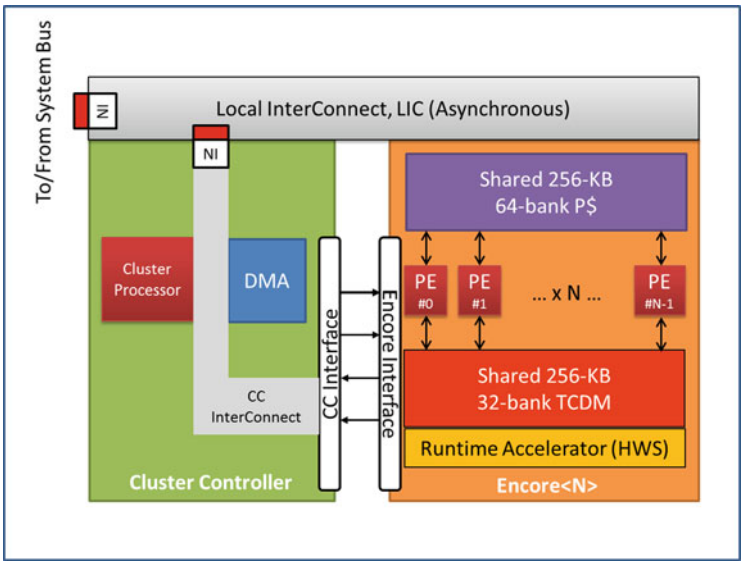

Fig. 3.2 Detailed cluster template

In conclusion, in order to program one cluster, one has to provide code for the 17 cores and to program carefully the DMA and the HWS to manage synchronizations and memory transfers.

To program the STHORM platform, an OpenCL support has been developed by ST. This implementation can be used to overcome the complexity of programming such a complex many-core platform. This OpenCL implementation is built on top of a low-level software layer called the resident runtime. It handles the execution of one cluster: it exposes a hardware abstraction layer of the programmable components of the cluster (DMA, HWS, memory allocators) and also a really simple set of services to launch software tasks on the PEs. This SW layer is resident, meaning that the corresponding code is always loaded in the fabric. These SW layers can be executed by several virtual platforms provided by ST. These virtual platforms can execute either native x86 code or target ARM code for the host and STxp70 code for the PEs [1]. Programmers have therefore the choice between a coarse-grain programming model or a fine-grain set of services that can be used to program the fabric by hand.

Within this context, there is a need for a homogeneous programming model and a uniform semantics for both the host side and the fabric side. This SW layer should also be able to support efficient code generation and easily integrate SW generation flows. It should also be configurable and able to capture fine-grain design choices, with low footprint. The emerging multicore communication API standard (MCAPI) [2, 3] has been identified as a good candidate which can meet all the previous requirements and can be used as an intermediate representation for the SMECY tool chains.

3.3 SystemC/TLM Platform

In this section, we describe the SystemC/TLM virtual view of the P2012 IP design. First, we will present the platform components, then the simulator characteristics. As a reference, we will define the available simulation environment. This simulator is part of the P2012 Software Development Kit.

3.3.1 Platform Components

The simulation platform is composed of a host, two memories, the P2012 IP model, and convenient blocks for platform services called "test bench driver" (see Fig. 3.3).

Two flavors of host are available: POSIX and native wrapper. The first one emulates host core behavior by running host driver in a separated POSIX thread. The second POSIX thread runs the SystemC kernel, both of them communicating using Unix pipes. The native wrapper is a standard SystemC module. It runs code using HCE (host code emulation) technology, i.e., code and variables are located in the platform memory model. The host selection is done using simulation options.

The P2012 IP model is composed of a fabric controller, a fabric, and optionally a memory. The aim of the fabric controller is to boot the fabric, to ensure resource management, and to fire interrupts to the host processor if needed.

The fabric is composed of one or several clusters. Each of them can be either homogeneous or heterogeneous, i.e., respectively, don't or do embed specific hardware assist design. A cluster is composed of an array of STxP70 processors, a peripheral controller which allows to drive core pins (boot address, interrupt pins, reset, fetch enable) and which embeds a timer and a mailbox, one or several DMAs with an optional connection to a stream interconnect for hardware processing elements feeding, a HWS to mainly speed up software barriers.

Additional specific hardware processing elements (see Fig. 3.4) can be plugged to one or several clusters. A dynamic library is loaded and runs the HWPE scenario.

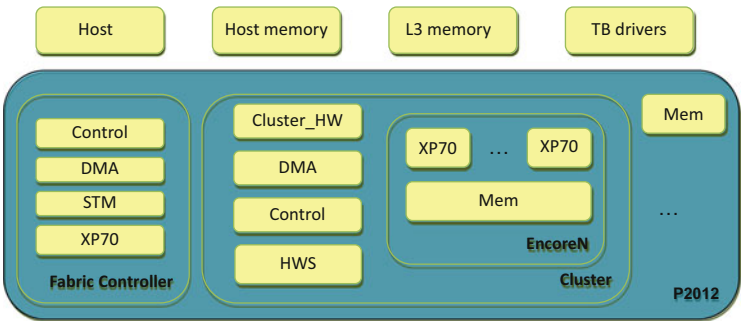

Fig. 3.3 SystemC/TLM platform

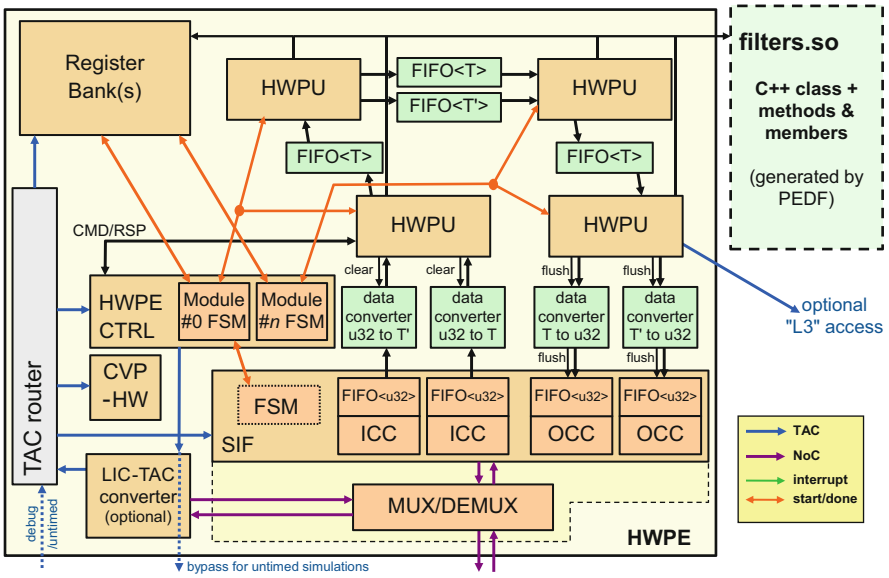

Fig. 3.4 HWPE block diagram

The platform has the capability to adapt its architecture following HWPEs structure (number of HWPEs, number of input and output ports, a register bank if any, backdoor ports).

The platform is delivered with a set of integration test examples to help the user to configure, build, and launch the platform.

3.3.2 Simulator Characteristics

The simulator is based on the SystemC 2.2 kernel. It uses STMicroelectronics Transaction Level Model C++ code. At the time of writing this chapter, only 32 bit binary on Linux RedHat 5 is supported.

This platform is highly configurable. The architecture is defined by an XML file. Typically, we can define if a fabric controller stands or not, how many clusters are in the fabric, if each cluster is homogeneous or not, how many STxP70 cores or DMA are contained in a cluster, etc.

The model is accurate compared to the design, i.e., produces the same output data from same STxP70 binary and data inputs. Moreover, the simulation is reproducible, i.e., execution order and simulation time are strictly identical (native wrapper host only). This is mandatory to fix tricky bugs.

3.4 Cosimulation Platforms

A cosimulation platform is a SystemC model where both C++ and HDL models stand. The following section presents several kinds of cosimulation platforms available for P2012.

3.4.1 IP Level Cosimulation Platform

In this case, the system on chip backbone is a TLM, the whole P2012 chip in HDL (see Fig. 3.5).

This platform can be either a 32-bit, or 64-bit binary. This later is required for long traces record. At the time of writing this chapter, only Cadence simulator is supported.

3.4.2 HWPE Level Cosimulation Platform

We have developed an HWPE level cosimulation platform in order to speedup the simulation and also to have the capability to start HWPE HDL debug while other HWPEs are not yet available in HDL. Using configuration parameters, we can cosimulate one, several, or all HWPEs (Fig. 3.6).

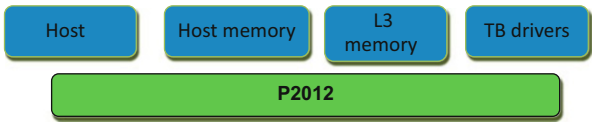

Fig. 3.5 IP level cosimulation

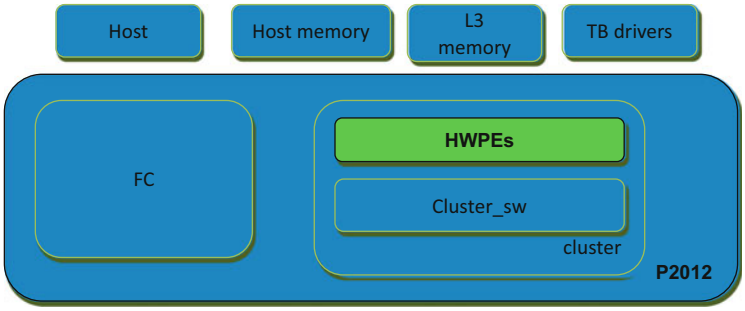

Fig. 3.6 HWPE level cosimulation

3.5 Fast Simulation Platform

This section presents the virtual platform used in the STHORM [1] tool chain for software development. This platform is called Gepop and it is written in C programming language and aims at providing a fast and timing accurate simulation environment. The instruction set architecture of the STxP70-4 processors is simulated using a classic ISS (instruction set simulator) returning the execution time per instruction to model the latency. This platform models only what is visible from the simulated software in terms of functionalities, performance, and power consumption in a way to provide fast simulations with a reasonable accuracy loss. Experiments indicate that an average timing precision of 90% can be achieved with a small overhead on simulation speed, whereas going above 95% often implies significant overheads. An accurate SystemC/TLM platform based on the loosely timed (LT) model has also been developed for STHORM, but mostly for verification purpose (integration tests and RTL cosimulation). This platform cannot be used for software development due to too slow simulation. The speedup between the virtual platform presented in this section and the SystemC/TLM platform has been measured between 5 and 20 depending on use cases, even when the SystemC/TLM platform does not provide any performance estimation.

This platform provides following features. First, a plugin mechanism: in general, each plugin represents a functional unit such as a core and a DMA. Plugins can be dynamically loaded and instantiated several times to match the actual architecture specification. A plugin also provides and requires interfaces to handle connection mechanisms. An interface is a set of methods that allows a plugin to communicate with other plugins. Using the interface mechanisms, functional modeling of the communication occurring in the real hardware can be achieved. The connection between two interfaces is based on unique string identifiers, like in service-oriented architectures. There are typically plugins for cores, memories, interconnection networks, and DMAs.

The second feature is a fast thread scheduler. A thread is used to model the functional behavior of unit responsible for a hardware activity, e.g., a processor sending requests to an interconnection network. A plugin can create several threads if it can have several activities in parallel. A thread can be active or inactive in order to represent the status of the activity. The scheduler is then in charge to manage all the active threads. For example, the plugin for the ISS is creating one thread per core. The thread is active when the core can execute instructions and it is inactive when the core is stalled due to the pipeline or while waiting for a transaction request completion.

The last feature is the timing estimation: each thread has its own time counter which is increased when it is simulating activity, e.g., a processor instruction. Then, when a time counter is updated, the scheduler is called in order to schedule the thread which has the lowest time count.

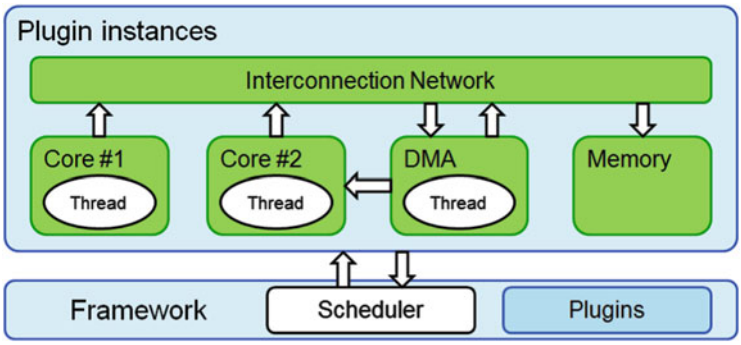

Fig. 3.7 Overview of the virtual platform used for software development

An overview of the typical use of the virtual platform and the framework associated with it is illustrated in Fig. 3.7. In general, each core has its own plugin with a thread generating requests to other devices. To do so, it is connected to a plugin modeling the interconnection network. Then, the network is connected to plugins that can receive memory-mapped operations such as a memory and a DMA. These plugins may also be connected to other plugins. For example, the DMA can send interrupts to cores or send memory transfer requests to the network. The power estimator is integrated in this environment.

3.6 Simulation Platform Usage

The simulation platforms can be used to simulate the behavior of the complete STHORM platform, both host coprocessor and fabric accelerator. To enable fast application simulation to ease complex development on the STHORM platform, several execution modes are offered to the users: the basic idea is to offer a large range of operating mode for the simulation, trying to offer good alternatives to cope with the accuracy versus performance trade-off.

Figure 3.8 shows the various possibilities offered by the simulation platforms. The SystemC/TLM platform is mostly used for hardware verification. The Gepop platform can be used to simulate applications running on top of the STHORM platform.

The Gepop platform can run application code either using regular x86 code encapsulated in POSIX threads or native code simulated by ISS. Depending on the accuracy that is desired by the user the execution mode can be set independently between host and fabric.

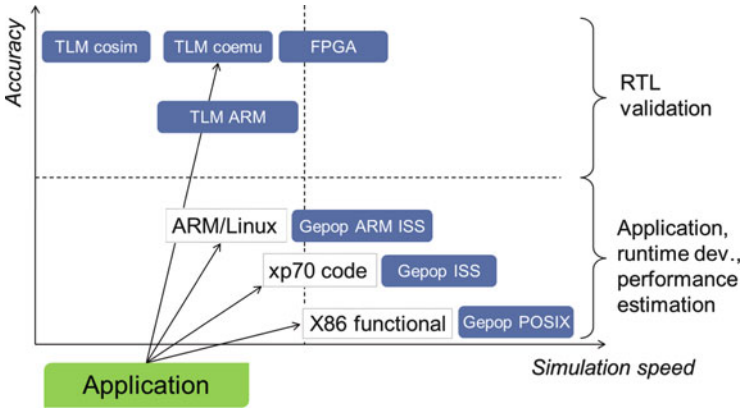

Fig. 3.8 Simulation modes and their mode of operating

References

1. D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1137–1142. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228568
2. The Multicore Association, "Industry standards to solve multicore challenges," http://www.multicore-association.org/, 2011.
3. The Multicore Association, "Multicore Communication APIs," http://www.multicore-association.org/workgroup/mcapi.php, 2011.

Chapter 4
The Architecture and the Technology Characterization of an FPGA-Based Customizable Application-Specific Vector Coprocessor (ASVP)

Roman Bartosiński, Martin Daněk, Leoš Kafka, Lukáš Kohout, and Jaroslav Sýkora

4.1 Introduction

"The job of a computer architect is to build a bridge between what can be effectively built and what can be programmed effectively so that in the end application performance is optimized" [1]. Indeed, in the last decade we have seen a wide deployment of parallel architectures in the form of chip-level scalar general-purpose multiprocessors (CMP) and streaming processors (GPU), but this was not met with a generally accepted solution to the problem of programming these systems in some unified manner. Examples of the programming interfaces include OpenMP, MPI (for CMPs), and OpenCL, CUDA (for GPUs). Thus we see that a compute architecture has to be designed in such a way to allow an efficient programming and applications development.

In the following text we propose an architecture that satisfies the following requirements:

- As the targeted technology is FPGA, the architecture has to be customizable for different applications to take advantage of the reconfigurability.
- The architecture has to support both floating-point (long-latency) and integer (short-latency) operations found in DSP and video applications. The long-latency FP operations, combined with the common need of DSP applications to support vector reductions, pose some new challenges.
- Several hardware technology nodes of different characteristics should be supported without a need for manual software changes in the firmware. Specifically, the firmware programmer should be shielded from the impact of adapting

R. Bartosiński • M. Daněk (✉) • L. Kafka • L. Kohout • J. Sýkora
UTIA AV CR, v.v.i., Pod Vodarenskou vezi 1143/4, Praha 8, 182 08, Czech Republic
e-mail: bartosr@centrum.cz; xdanek@email.cz; leos.kafka@centrum.cz; kohoutl@utia.cas.cz; sykora@utia.cas.cz

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_4,
© Springer Science+Business Media New York 2014

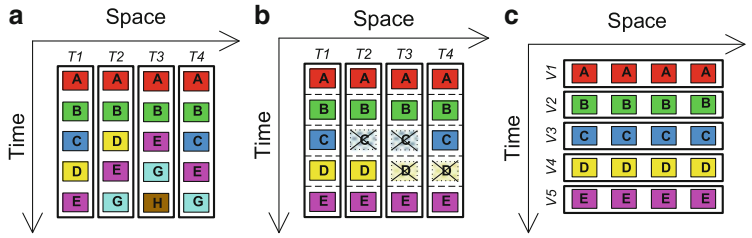

Fig. 4.1 Thread-based vs. vector-based execution of a data-parallel computation. Note that in practice, threads are usually executed in SIMD clusters (**a**) Threads (**b**) SIMD (**c**) Vectors

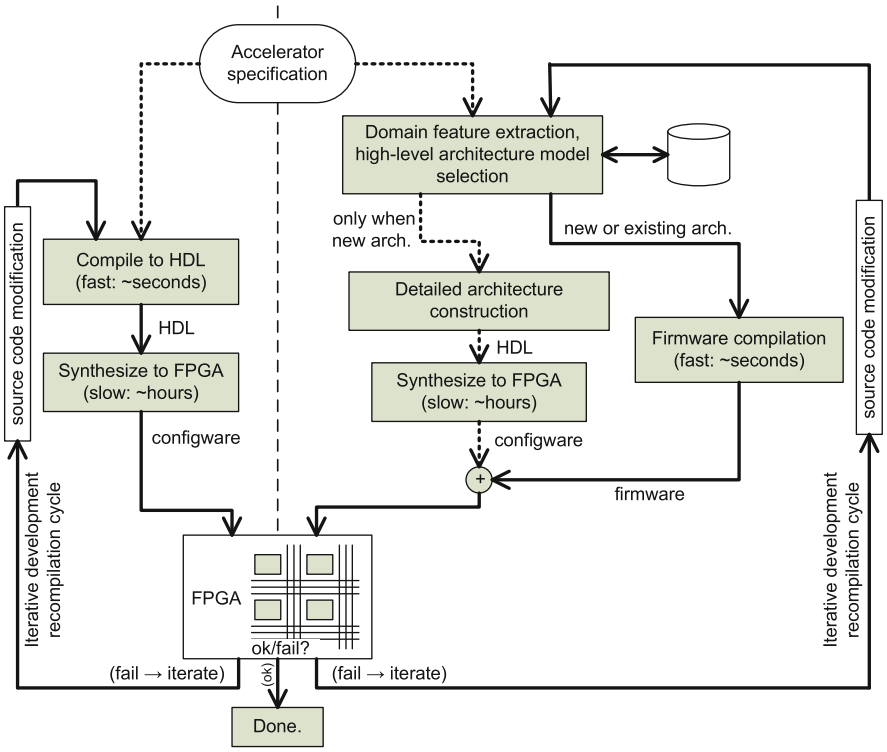

Fig. 4.2 Custom accelerators development work-flow: (**a**) traditional (*left-part*); (**b**) proposed (all). *Dashed lines* indicate processes run once; *full lines* mark iterative paths

the latency/frequency ratio of the hardware units to the target technology (Figs. 4.1 and 4.2).

Clearly, given the reconfigurable FPGA implementation technology that allows very high degree of hardware customization for a given application, and the desire to pack as many instances of the execution cores in the chip to maximize performance, the proposed compute architecture cannot rely on hardware-assisted dynamic

parallelism extraction. Instead all the parallelism has to be explicit beforehand, and the burden is placed on the compiler and/or programmer.

The targeted application areas (DSP, video) are highly data parallel. The question is how much is the parallelism "regular" or "irregular" in terms of memory access patterns and operations performed by different data-parallel strands. Two architecture organizations exist to deal with it: (a) *thread-based* (with SIMD) and (b) *vector-based*. For data-parallel applications the threading architectures are usually implemented using SIMD clusters (in GPUs) using. Wider SIMD (more threads executed simultaneously in lockstep) improves not only the area efficiency but also the penalty for diverging threads. Irregular data-parallel applications are more readily implementable on a thread-based architecture than on a vector-based one, because in the later diverging data-parallel strands have to be dealt with by explicit masking and selection of vector elements. Therefore, contemporary high-performance number-crunching ASICs (GPUs) are based on threads.

The disadvantage of threading architectures (compared to the vector-based ones) is that they require a very rapid issue of fine-grained instructions to control the execution pipeline. The instruction fetch/decode/schedule circuitry can be resource costly, thus the use of SIMD clusters. Performance is sensitive to the static schedule of the instructions, thus an optimizing compiler from a higher-level language is required.

However, developing the optimizing compiler for a highly customizable architecture, which can be tailored and tuned for each application thanks to the reconfigurable nature of the target technology (FPGA), can be very difficult.

4.2 Related Work

Vector processing was shown to be a good match for embedded applications. In [2] the VIRAM vector processor was evaluated using the EEMBC multimedia benchmark suite, and it was found, for example, to outperform VLIW processors by a factor of 10. The original VIRAM chip is 0.180 μm custom design clocked at 200 MHz, 1.6 GFLOP single-precision performance, with an on-chip 13 MB 8-bank embedded DRAM (a crucial feature). The large multibanked local memory is used as a software-controlled staging buffer that balances the latency/bandwidth ratio of the memory hierarchy.

A centralized vector register file (VRF) cannot be scaled to support many chaining functional units, as each FU requires additional ports to the VRF. CODE [3] vector micro-architecture proposes to partition the architectural VRF into several physical VRFs distributed in clusters. A renaming table is used to keep track of the location of architectural registers, and a history buffer is employed to support precise exceptions.

SCALE [4] is an implementation of a vector-thread architecture in a custom 0.180 μm design, clocked at 400 MHz, with four vector-thread (VT) lanes. Vector lanes can either all execute the same instruction (as in a traditional vector processor),

or each VT lane can fetch and execute different *Atomic Instruction Blocks* concurrently to the other lanes, allowing true multi-threaded operation. Similar to CODE, lanes are partitioned into clusters; however, this partitioning is explicit and visible in ISA.

Following the success of VIRAM, several vector processors were implemented in FPGAs using the VIRAM ISA. Micro-architectures of these processors are usually traditional and not so advanced as in CODE or SCALE. VESPA [5] is design-time configurable softcore implemented in Stratix-I and Stratix-III FPGA. The processor allows ISA subsetting, and some of the primary design parameters that affect both performance and resource usage can be configured (e.g., the number of vector lanes, vector lane width, memory crossbar).

VIPERS [6] is similar to VESPA, but it is less strict to VIRAM ISA compliance, and more tailored to the FPGA target technology. It offers a few new instructions to take advantage of the MAC (multiply-accumulate) and BRAM units in FPGAs. The new MAC instructions perform integer summation (reduction) within vector registers. The processor can be also augmented with a fast vector-lane-local memory that can accelerate table lookup functions.

FPVC [7] is an FPGA-based vector coprocessor with floating-point units, implemented in Virtex 5. The processor has unified scalar and VRF.

4.3 Description

4.3.1 Hierarchy Layering

The system-level view is presented in Fig. 4.3. Similar to streaming architectures (Imagine [8], Cell [1]), the execution control is hierarchical and structured into three layers. (1) Task scheduling is dedicated to the host CPU. In embedded solutions with a simple and/or static task scheduling, this layer may be omitted and the scheduling process distributed among the BCE cores. The inter-BCE synchronization is handled by the Communications Backplane (δ). (2) Scheduling of the vector instructions is realized in the control MCU imbedded in each BCE core. The MCU forms and issues wide instruction words (α) to the Vector Processing Unit. In the implementation we used the 8-bit Xilinx PicoBlaze processor. A sample structure of the instruction word is given in Fig. 4.4; the field bit-widths depend on the configuration (on the number and size of local memories and address generators). (3) Data path multiplexing and vector processing is realized in the Vector Processing Unit. The unit handles both the vector-linear and vector-reduction operations, as well as local memory banks access scheduling (β).

The memory hierarchy is exposed on two levels: (1) Global off-chip shared memory is accessed through the Streaming DMA engine. In the Xilinx technology the engine uses the MPMC (Multi-Port Memory Controller) and NPI (Native Port Interface). The engine is programmed by the MCU in each BCE core (γ), and

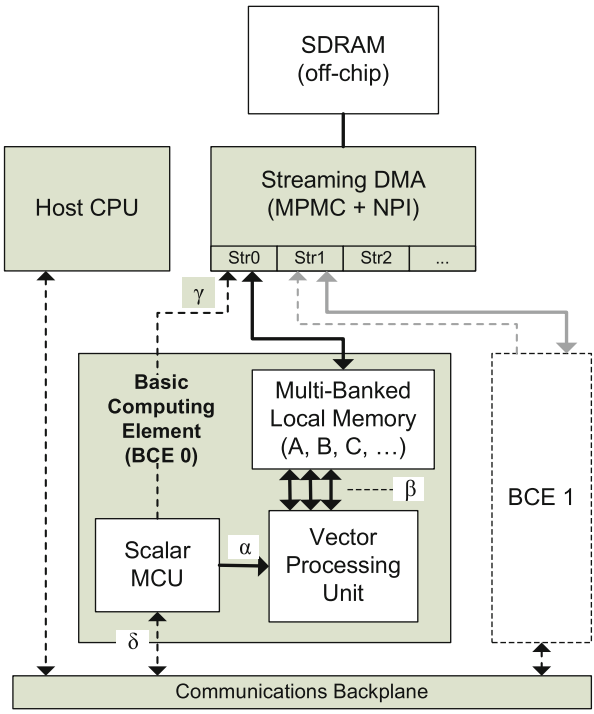

Fig. 4.3 System-level organization

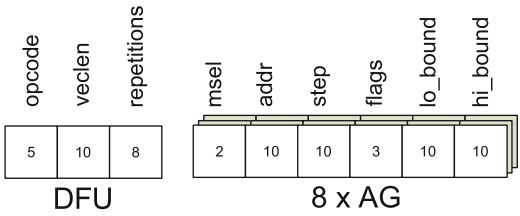

Fig. 4.4 Structure of the VLIW-encoded vector instruction (α). The field bit-widths correspond to the BCE configuration with four memory banks (2-bit "msel"), each with 1024 words (10-bit address fields)

delivers data into the BCE local memory banks. (2) Local storage in each BCE is realized using multiple memory banks (BlockRAMs in FPGA). The Vector Processing Unit (VPU) can access all the banks in parallel (β).

The on-chip local storage is used as the working set staging buffer. A kernel function running in the BCE accesses data with non-unit strides, and often the same data is reused multiple times in one computation run (temporal locality). In contrast, off-chip memory (DDR DRAM) has high latency and it delivers high bandwidth only when unit-strided long data arrays are transferred.

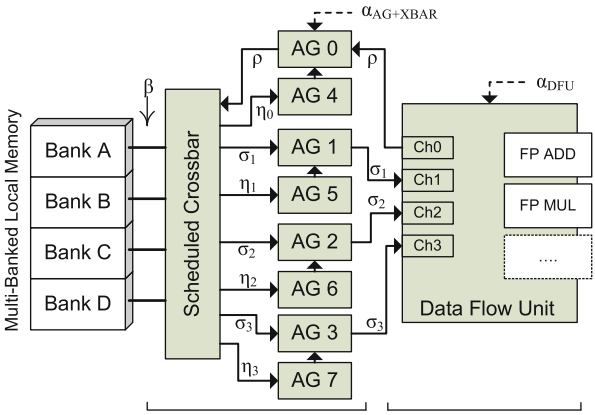

Fig. 4.5 Canonical structure of the vector processing unit with local memory banks

4.3.2 Vector Processing Unit

Figure 4.5 shows the canonical structure of the VPU. The VPU fetches data vectors from the local memory banks, processes them, and stores the result back. The memory banks are dual-ported: one port of each bank is connected to VPU, the other to the Streaming DMA engine. Thus it is possible to overlap computations and data transfers in the same banks.

The architecture does not contain a traditional VRF. Register files are notoriously cumbersome and inefficient to implement in FPGA: even the simplest 3-port (2R-1W) register file requires duplicate (data-redundant) dual-ported BlockRAMs. In place of a VRF we employ a multibanked local store with a crossbar. In the default configuration with four memory banks this allows to read/write up to four values at a time. Further, the local memory banks are not statically partitioned into architectural vector registers. Instead, an application is free to partition the available memory into vector variables: some applications take advantage of a few very long vectors (e.g., FIR); others prefer a lot of shorter vectors to implement complex computation (e.g., IMGSEG).

In VPU data is processed in three layers. The vector computation itself takes place in the Data-Flow Unit (DFU). The DFU has several input channels (generally two to three) and one output channel. The channels are modeled as FIFO queues. The addressing of vectors in the data banks is handled by a layer of Address Generators (AG 0–7). Address Generators produce streams of addresses that are routed through the crossbar layer into the memory banks. In the case of operand reads, the data extracted from the memory banks is routed back through the layers into the DFU input channels.

It is an important design decision that the DFU is connected to the other parts of the VPU only through the data FIFO queues (Channels 0–3). First, this ensures that the vector instruction set is completely orthogonal: any addressing mode can be used in any instruction. Second, as the DFU is the most application-specific

(customizable) part of the architecture, its clear and simple separation from the other units eases the hardware development. And finally, the asynchronous[1] FIFO interface physically decouples DFU and AG so that timing and routing constraints in the FPGA synthesis and implementation process are met.

4.3.3 Space-Time Scheduled Crossbar

Several vector operands of an instruction *can* lie in the same memory bank. The crossbar automatically handles the time-domain access scheduling when the requests from the Address Generators cannot be satisfied by the switch matrix in the space domain. For each memory bank the crossbar schedules accesses independently from the other banks. Then, considering one memory bank alone, a bit mask vector of Address Generators that request access to the particular bank is constructed. (The bit mask is constant for each vector operation.) Bank schedule (i.e., which AG is granted access to the given bank in a particular clock cycle) is determined by a circular shift register into which a single hot one is loaded at the beginning. Individual stages (flip-flops) of the shift register are bypassed when the corresponding bit of the request bit mask is zero. This way the length of the circular shift register is dynamically adjusted to cover exactly the AGs that request access to the bank. For example when AGs 2, 3, and 6 request access to Bank A, the req. bit mask for the bank is "00110010," outputs of the shift register will be ("00100000," "00010000," "00000010," ...), thus the schedule is (2, 3, 6, 2, 3, ...).

4.3.4 Address Generators

In the proposed architecture there is no traditional VRF. Instead all working data is stored in local memory banks. In the default configuration each bank is a flat array of 1024 32-bit words, suitable for holding floating-point values. The banks are efficiently implemented using the embedded memory blocks found in all modern FPGAs (BlockRAMs in Xilinx FPGAs).

Vectors are extracted from the memory banks by the Address Generators (AGs). Each operand of a vector operation (input or output) has to completely lie in a single memory bank. The full hardware configuration of the VPU sports two AGs for each operand channel. The main AGs, denoted 0–3 in Fig. 4.5, handle basic addressing modes: linear or strided ($increment \neq 1$) access (the increment can be negative), with lower and upper wraparound bounds (overflowing the upper bound resets pointer to the lower bound, and vice versa).

[1] Asynchronous with respect to a cycle count an operation may take.

The second set of AGs 4–7 is for indexed accesses. These AGs have the same configuration registers as the main AGs, but the data stream (η_{0-3}) they read from the bank memories is passed on to the corresponding main AG. There it is used as indices added to the local addresses being sent down to the memory bank. Main AG can be configured to increment its local pointer only when its slave indexing AG reaches upper/lower bound; this is useful when executing a batch of several vector operations in the DFU. Slave indexing AG can read indices from a memory bank, or it can short-circuit its address pointer directly to its output; this is useful when the index stream is regular and it can be computed directly.

4.3.5 Data-Flow Unit

DFU operation is controlled by three fields from the vector instruction word α: (1) operation code, (2) vector length, and (3) number of repetitions. The "number of repetitions" field allows to automatically restart the same operation over multiple times to create a "batch" of operations. Obviously, this trick lowers the total number of distinct vector instructions that must be issued by the MCU, thus lowering its load. More importantly, however, Address Generators are *not* rewound in between the operation restarts. Thus by properly setting AG op-code fields it is possible, for example, to compute one result row in a matrix multiplication using single DFU instruction. Figure 4.6 shows how this is done, and Fig. 4.7 lists the C source code that is executed on the MCU.

4.3.6 Programming and Simulation

Figure 4.9 shows the software flow. There are two distinct instruction sets in the architecture. The control MCU executes classical scalar ISA and it is programmed in the C language. The architecture currently uses the 8-bit PicoBlaze processor as the MCU, and we have developed an optimizing C compiler in the LLVM framework for the PicoBlaze target. The VPU executes VLIW-style vector instructions (denoted α in figures). The vector instructions are prepared by MCU in a so-called "instruction forming buffer"; the forming buffer is an I/O periphery of the MCU. The MCU is programmed in C, and functions such as `pb2dfu_set_fulladdr()` or `pb2dfu_restart_op()` are used to write into the buffer. Currently the vectorization of an application to a sequence of vector instructions has to be performed manually.

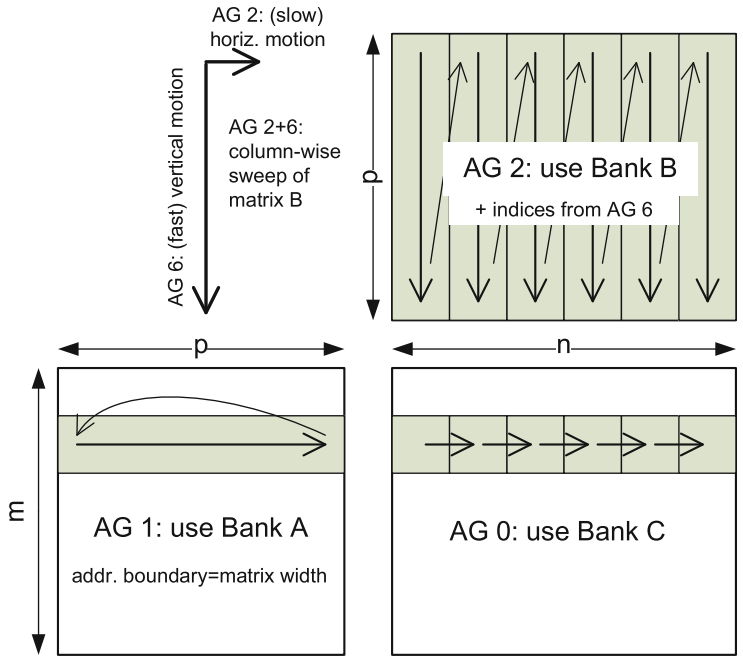

Fig. 4.6 Using batches and AG features, one result row in matrix multiplication can be computed in a go

4.4 Customization Methodology

The DFU, which realizes the actual compute operations, is the primary target of the application-specific customization effort. In general, three kinds of operations on vectors are recognized: (a) element-wise function mapping (e.g., vector addition), (b) full reductions (e.g., vector summation), and (c) prefix reductions (e.g., cumulative summation). To achieve good performance the low-level hardware compute units, such as a floating-point adder or multiplier, are pipelined. For example, depending on technology, the latency k of the FP-Adder is between 3 and 6 cycles. Pipelining of the element-wise operation is trivial; after the first k cycles, one result is obtained in each cycle.

Full reduction r of vector A_i using operator $\circ$ can be defined as

$$r = A_0 \circ A_1 \circ \ldots \circ A_{n-1} = \overset{n-1}{\underset{i=0}{\Psi_\circ}} A_i \tag{4.1}$$

(e.g., summation is $\Psi_+ \equiv \sum$; $r = \sum_{i=0}^{n-1} A_i$).

To pipeline the reduction the operator $\circ$ (with latency $k_\circ$) has to be associative and commutative, and there has to exist a neutral element $\epsilon_\circ$ (e.g. $\epsilon_+ = 0$). Then we can write (for the case $k_\circ = 2$):

Fig. 4.7 Matrix multiplication MCU firmware source in C

```
/* C[m][n] := A[m][p]  * B[p][n]  */
void matmul(unsigned char m, unsigned char n,
            unsigned char p)
{
  unsigned char a = 0, c = 0;

  /* set common parameters*/
  pb2dfu_set_cnt(p); // DPROD ofl ength p
  pb2dfu_set_repetitions(n);
  pb2dfu_set_inc(DFUAG_0, 1);          // C
  pb2dfu_set_inc(DFUAG_1, 1);          // A

  /* B (AG2): all columns */
  pb2dfu_set_fulladdr(DFUAG_2, MBANK_B, 0);
  pb2dfu_set_inc(DFUAG_2, 1);          // B
      // Use index from DFUAG_IDX_2;
      // Step only on DFUAG_IDX_2 overflowing.
  pb2dfu_set_agflags(DFUAG_2,
      AGFL_USE_IDX | AGFL_STEP_IDXBND);

  /* IDXAG2 (= AG 6): As number generator. */
  pb2dfu_set_agflags(DFUAG_IDX_2, AGFL_NUMBGEN);
  pb2dfu_set_inc(DFUAG_IDX_2, n);       // B indices
  pb2dfu_set_addr(DFUAG_IDX_2, 0);
  pb2dfu_set_bound_addr(DFUAG_IDX_2, 0, (p-1)*n);

  for (unsigned char mm = 0; mm < m; ++mm) {
    /* A (AG1): row m that starts on address
       'a' and stops at a+p-1 */
    pb2dfu_set_fulladdr(DFUAG_1, MBANK_A, a);
    pb2dfu_set_bound_addr(DFUAG_1, a, a + p- 1);

    /* C (AG0): row m that start on address
       'c' in bank C */
    pb2dfu_set_fulladdr(DFUAG_0, MBANK_C, c);

    /* wait for previous operation to complete*/
    if (mm) pb2dfu_wait4hw();

    /* dot-product row A[a][*] and all columns
        of B[*][*] to a row C[a][*] */
    pb2dfu_restart_op(DFU_DPROD);

    a += p;  c += n;
  }
  pb2dfu_wait4hw();
}
```

$$r = \epsilon_o \circ A_0 \circ A_1 \circ \ldots \circ A_{n-1} = \tag{4.2}$$

$$= (\epsilon_o \circ A_0 \circ A_1) \circ (\epsilon_o \circ A_2 \circ A_3) \circ \ldots = \tag{4.3}$$

$$= (\epsilon_o \circ A_0 \circ A_2 \circ \ldots) \circ (\epsilon_o \circ A_1 \circ A_3 \circ \ldots) \tag{4.4}$$

The last form leads to a hardware implementation using the circuit in Fig. 4.10. The reduction computation in the circuit goes through three phases: (1) In the first k_o cycles the operator output c is invalid; thus the input b must be supplied with the

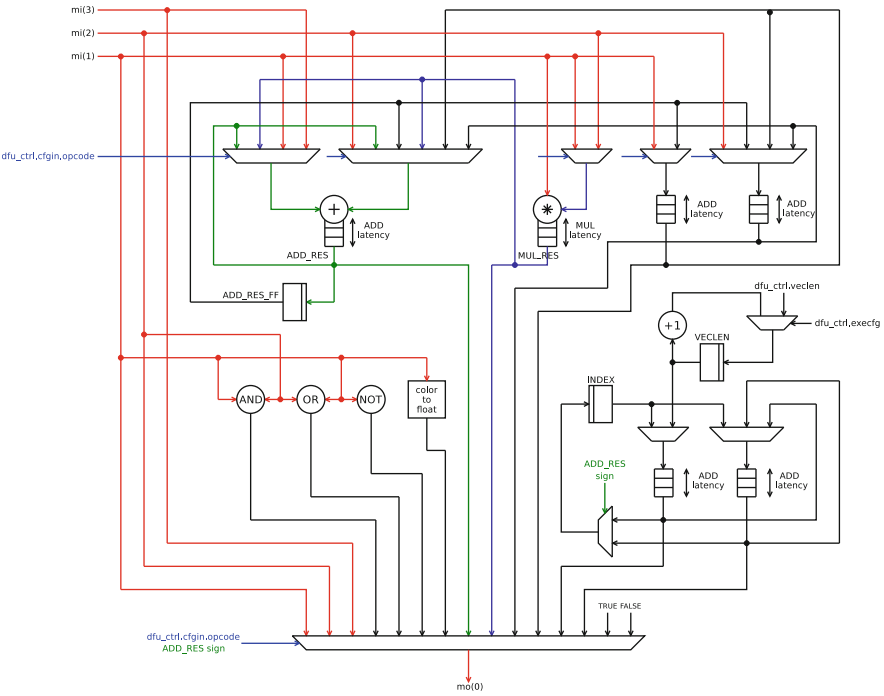

Fig. 4.8 Simplified internal structure of the DFU implemented for the *Image Segmentation* application. The control FSM is not shown

value $\epsilon_\circ$. (2) In the next $n - k_\circ$ cycles the rest of the vector A_i is consumed, and partial results on c are routed to the input b. (3) In the windup phase, which lasts w cycles, the partial results circling in the pipeline are gradually reduced down to the final value r.

In the windup phase it takes $k_\circ$ cycles (one round) to halve the number of values circling in the pipeline, thus $\log_2 k_\circ$ rounds is required:

$$w = (1 + \log_2 k_\circ)k_\circ - 1 \tag{4.5}$$

(The last equation is tabulated in Table 4.1.) Therefore, the pipelined (parallel) implementation of the full reduction is much faster that the sequential one:

$$t_{seq,\Psi_\circ}(n) = k_\circ \cdot n \tag{4.6}$$

$$t_{par,\Psi_\circ}(n) = n + w = n + (1 + \log_2 k_\circ)k_\circ - 1 \tag{4.7}$$

The execution times t are given in clock cycles here. In the pipelined algorithm the windup latency is w cycles, but there are only $k_\circ$ values in the execution pipeline at the beginning. Hence, the operator $\circ$ is used only in $k_\circ - 1$ cycles, and empty cycles are inserted into the pipelined unit otherwise.

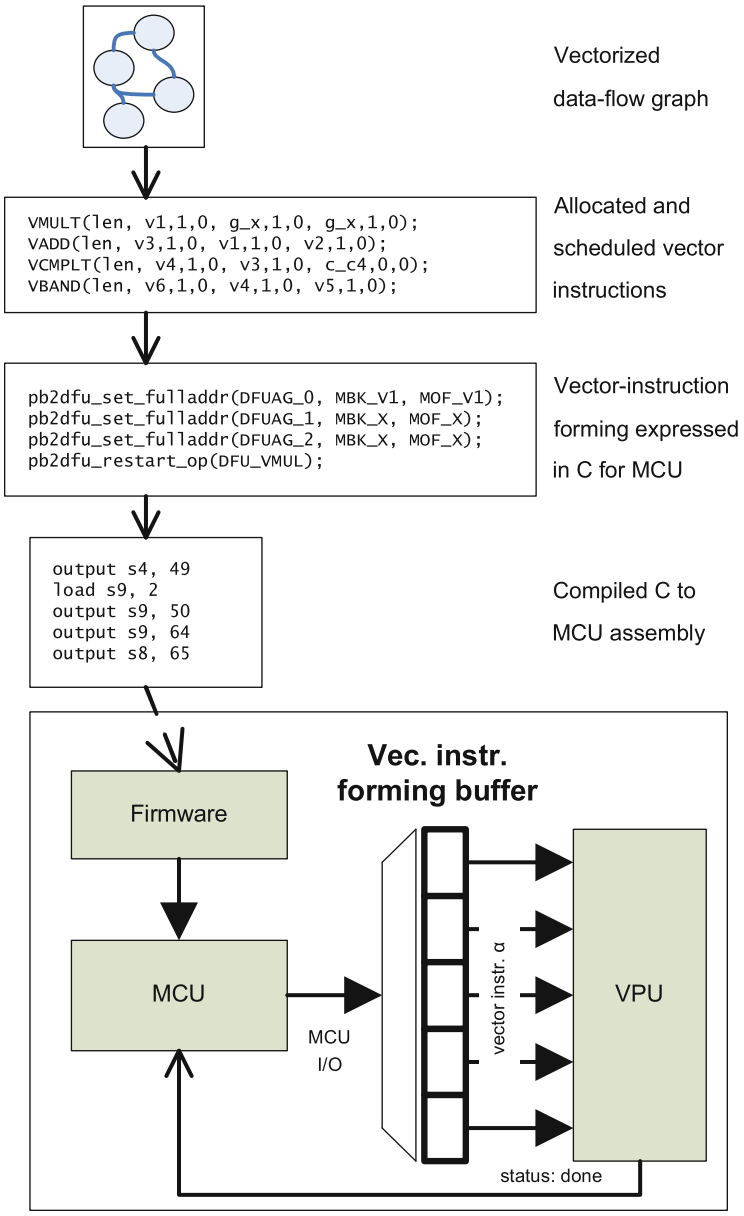

Fig. 4.9 Software programming flow

Table 4.1 Tabulated values of Eq. 4.5: reduction windup cycle count w vs. latency of a compute unit $k_\circ$

$k_\circ$	1	2	3	4	5	6	7	8
w	0	3	7	11	16	21	26	31

Prefix reduction $\Xi_\circ$ of a vector A_i into vector C_j is defined:

$$\underset{j=0}{\overset{n-1}{\forall}} : C_j = \overset{j}{\underset{i=0}{\Psi_\circ}} A_i \quad ; \quad C = \overset{n-1}{\underset{i=0}{\Xi_\circ}} A_i \tag{4.8}$$

Prefix reductions are much more difficult to implement in a pipelined manner. The pipelined algorithm requires $\log_2 n$ passes over the array, and in each pass i roughly $(n - 2^i)$ operations are performed that can be pipelined. Thus:

$$t_{seq,\Xi_\circ}(n) = k_\circ \cdot n \tag{4.9}$$

$$t_{par,\Xi_\circ}(n) \approx \sum_{i=0}^{\log_2 n} (n - 2^i) = \tag{4.10}$$

$$= n \cdot \log_2 n - n + 1 \tag{4.11}$$

The parallel algorithm is faster for some combinations of $k_\circ$ and n, but it is asymptotically slower (in n) than the sequential algorithm ($O(n) < O(n \cdot \log_2 n)$) when only one instance of the reduction operator $\circ$ is available. Thus, when the prefix reduction algorithm is needed, we use the sequential implementation.

4.4.1 Example

In the IMGSEG application there is an operation that locates the minimal (maximal) value in a given vector of floating-point numbers. The operation either returns the value (then it is called VMIN, VMAX), or the integer index where the value is located (INDEXMIN, INDEXMAX).

These operations can be efficiently implemented as full reductions. First, define scalar function $Min(a, b)$ that simply returns the lesser of the two arguments. The function is commutative ($Min(a, b) = Min(b, a)$) and associative ($Min(a, Min(b, c)) = Min(Min(a, b), c)$). The neutral element is $\epsilon_{Min} = +\infty$ because $Min(a, +\infty) = a$. Thus we can place $VMIN \equiv \Psi_{Min}$. The Min function can be implemented in hardware as shown in Fig. 4.8 using a floating-point subtraction unit and a multiplexer. The block is used in place of the general function $a \circ b$ in Fig. 4.11.

Another example of an application-specific vector instructions is VCONVR/G/B instructions. The instructions take a 32-bit pixel, extract a given 8-bit color

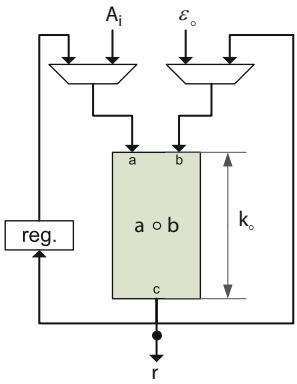

Fig. 4.10 Hardware model of the full-reduction $\Psi_\circ$

Table 4.2 Sample of operations implemented in the DFU

Operation	Type	Definition
VCOPY	M	$A_i \leftarrow B_i$
VADD	M	$A_i \leftarrow B_i + C_i$
VMUL	M	$A_i \leftarrow B_i \cdot C_i$
VMAC	M	$A_i \leftarrow B_i \cdot C_i + D_i$
VSUM	FR	$A_0 \leftarrow \Psi_+(B_i)$
DPROD	M+FR	$A_0 \leftarrow \Psi_+(B_i \cdot C_i)$
VMIN	FR	$A_0 \leftarrow \Psi_{Min}(B_i)$
INDEXMIN	FR	$A_0 \leftarrow Arg\{\Psi_{Min} B_i\}$
VCMPLT	M	$A_i \leftarrow (B_i < C_i)\ ?\ True : False$
VSELECT	M	$A_i \leftarrow (B_i \neq 0)\ ?\ C_i : D_i$
VCONVR	M	$A_i = int2float((B_i >> 16)\&0xFF)$

Type: M = element-wise function map, FR = full reduction

(R, G, or B) from it, and convert the color to a floating-point value. Using a conventional RISC vector ISA the operation would be implemented using at least three instructions (bit mask, shift, float conversion).

Table 4.2 lists some vector instructions we have implemented in DFU. Most of them operate on single-precision floating-point values. The DPROD operation is the dot-product that is very useful for implementing matrix multiplications. The VCMPLT operation compares two vectors element-wise and returns a vector of boolean values. The VSELECT operation is a vectorized conditional ternary operator from the C language.

When the list of operations for a given application is complete, the application-specific DFU is constructed to include the required data paths (Fig. 4.8). Currently this is done mostly manually in VHDL; however, it should be possible to synthesize the DFU automatically in a tool given a high-level specification such as the one in Table 4.2.

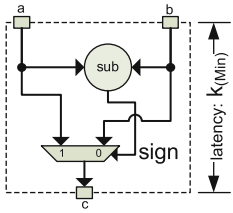

Fig. 4.11 Hardware data-flow realization of the $c = Min(a, b)$ function. Note that in hardware the subtraction operator may be pipelined depending on function (integer or float) and target technology and speed

4.5 Memory Interface Characterization

The memory interface is implemented using the standard MPMC supplied by Xilinx, extended with a front-end controller that accepts requests from individual VPUs and translates them to optimal NPI transactions. This section shows characterization of the memory interface. The experiments concentrate especially on MPMC configurations that utilize the NPI interface, since this interface is used as the primary interface to the MPMC module in the npi_if module.

4.5.1 MPMC Core in Virtex5

The first set of experiments evaluates the MPMC throughput through software simulation. The MPMC soft core was generated using the Xilinx XPS 12.3 tool. Transfer generators for the NPI interface were implemented through custom simulation models. A real DDR2 memory access was simulated through a model of DDR2 memory by Xilinx.

The configuration of the MPMC core was as follows:

- MPMC version 6.02.a
- 2x PLB 64b @ 100 MHz
- 2x NPI 64b @ 200 MHz
- Round-robin arbitration among user interfaces
- DDR2 64b @ 200 MHz

4.5.1.1 Simplex Operations at a Single NPI Port, XC5V

The first experiment measured throughput of a single NPI port. The goal was to evaluate maximal throughput of either type of data transfers (read or write) in the case when it is the only operation that is performed by the MPMC core at a time.

The ratio of the throughput of the NPI interface to a maximal theoretical throughput of the same interface was to be evaluated as well. The maximal theoretical throughput of a 64b interface running at 200 MHz is 1,600 MB/s.

Table 4.3 Throughput of a single NPI port for read and write operations of different length; Virtex5

Transfer length	Read [MB/s]		Write [MB/s]	
1 KB	1,274	80%	1,177	74%
4 KB	1,422	89%	1,330	83%
16 KB	1,464	92%	1,379	86%

An experimental workload consisting of data operations and batches that were defined is as follows:

- Read operation: an operation that fetches continuous chunk of data from the NPI interface. The operation is usually split into several bursts according to NPI interface constraints. Elapsed time is measured through a timer—it starts when the address of the first burst is issued and stops when the last data word is fetched from the NPI interface.
- Write operation: an operation that writes continuous chunk of data to the NPI interface. The operation is usually split into several bursts according to NPI interface constraints. Elapsed time is measured through a timer—it starts when the first data word is being written into the NPI interfaces and stops when the corresponding data buffer in the MPMC core is empty again.
- Batch: three write operations followed by three read operations. Note that each operation has to finish completely, i.e., the NPI interface has to be idle again, before the subsequent operation of the batch is started.
- Workload: four batches.

Three various lengths of transfer operations were considered—1 KB, 4 KB, and 16 KB. The length of burst at the NPI interface is equal to 32 data beats, which is equal to 256 bytes in turn. It means that the transfer of 1 KB is split to 4 bursts, transfer of 4 KB to 16 bursts, and the transfer of 16 KB to 64 bursts. The last one is close to a continuous data-flow through the NPI interface.

Results are shown in Table 4.3. The table shows NPI interface throughput in MB/s for read and write operations (columns 2 and 4, respectively) and ratio of these values to the maximal theoretical throughput of the NPI interface (columns 3 and 5).

The experiment showed that a continuous data-flow through a single NPI interface is close to 1,464 MB/s in case of read operations and 1,379 MB/s in case of write operations. These were equal to 92% and 86% of the maximal theoretical throughput respectively. The speed degradation in case of transfers of shorter lengths was caused by a latency of a pipelined NPI data path and the used time measurement technique—the timer was stopped after the last word has passed the MPMC internal buffer, but the NPI interface may be ready to accept another request long time before this event. Note that simplex operations only were considered in this experiment, but the NPI supports duplex operations.

Table 4.4 Throughput of two NPI ports for read and write operations of different lengths; Virtex5

	Read [MB/s]			Write [MB/s]		
Transfer length	NPI#0	NPI#1	DDR2	NPI#0	NPI#1	DDR2
1 KB	995	1,021	2,016	864	826	1,690
4 KB	1,015	1,052	2,067	841	813	1,654
16 KB	1,039	1,040	2,079	831	831	1,662

4.5.1.2 Simplex Operations at Two NPI Interfaces, XC5V

The second experiment measured throughput of the DDR2 interface that was fed by two NPI interfaces through the MPMC controller. As the maximal theoretical throughput of a couple of NPI interfaces (2×64b @ 200 MHz) is equal to the maximal theoretical throughput of the DDR2 interface (1×64b @ 200 MHz double data rate) and there is some overhead of a protocol at the DDR2 interface, it was expected that the two NPI ports could saturate a single DDR2 port, considering the MPMC configuration mentioned above. The goal of this experiment was to evaluate the throughput of the DDR2 interface in such a scenario.

The setting of the experiment is similar to the experiment shown in Sect. 4.5.1.1. The read/write operations, the batches, and the workload are defined identically. The only difference is that two NPI interfaces, instead of one, performed the same operation (almost) synchronously, i.e., either the read or write operation at both interfaces was issued at the same time.

Results are shown in Table 4.4. The table shows the NPI interface and DDR2 interface throughput in MB/s for read and write operations. It includes throughput of the first and the second NPI interfaces and the DDR2 interface for both read operations (columns 2, 3, and 4, respectively) and write operations (columns 5, 6, and 7, respectively).

The experiment showed that the continuous data-flow through the DDR2 interface that was fed by two NPI interfaces was 2,079 MB/s in case of read operations and 1,662 MB/s in case of write operations. The synchronization of two generators was not precise, so start time of the corresponding read/write operations differed slightly. On the other hand, this drawback was eliminated by a high number of bursts within single read/write operation.

A timing diagram of the experiment (16 KB) is shown in Fig. 4.12. The simulation time is shown on the X-axis and the transfer rate (i.e., the throughput) on the Y-axis. The value was put into the diagram at the time when the corresponding operation finished. The diagram shows four batches of six operations (three write operations followed by three read operations). Both NPI interfaces performed these operations almost synchronously.

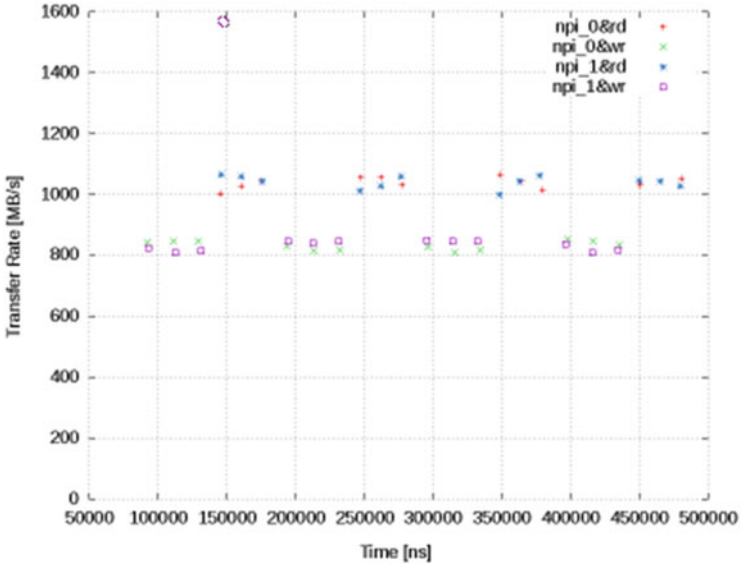

Fig. 4.12 Time diagram of simplex transfers issued from two NPI interfaces through MPMC core to a single DDR2 interface

4.5.1.3 Duplex Operations at Two NPI Interfaces, XC5V

The third experiment provided throughput of two NPI interfaces in case of duplex operations.

The NPI interface consists of read and write interfaces that operate independently and a shared address interface. Both read and write interfaces can operate concurrently. Although the maximal theoretical throughput is up to 3,200 MB/s in such case, the real throughput depends on burst lengths and throughput of the DDR2 interface.

An experimental workload consisted of data operations and batches that were defined as follows:

- Read and write operation: the operation that fetches continuous chunk of data from the NPI interface and writes continuous chunk of data to the NPI interface at the same time. Both read and write operations are split into several bursts and these bursts are interleaved. Elapsed time is measured through two timers.
- Batch: a sequence of read and write operations. It means that read and write operations are performed as fast and concurrently as possible and batch finishes when all these operations finish completely, i.e., the NPI interface is idle.
- Workload: four batches.

Table 4.5 Throughput of two NPI ports that perform duplex operations; Virtex5

	Read and write [MB/s]		
Transfer length	NPI#0	NPI#1	DDR2
8 KB, 8 KB	469 and 465	470 and 465	1,869
8 KB, 16 KB	619 and 613	38 and 38	1,308

Two combination of operation lengths within a batches were considered:

- 4×8 KB for both NPI interfaces
- 4×8 KB for the first NPI interface and 128×16B for the second NPI interface

The first one is used to evaluate an overall throughput of duplex data-flows issued through two NPI interfaces. The latter is used to evaluate throughput of a single data-flow issued from one NPI interface when large number of small random transfers is being issued at the second NPI interface in order to disturb the first data-flow. Note although the amount of data differs for both NPI interfaces (32 KB vs. 2 KB), they performed the same number of bursts (128).

Results are shown in Table 4.5. The table shows NPI interface and DDR2 interface throughput in MB/s for read and write operations.

The overall throughput of read and write operations in the case of two long data-flows (the "8 KB, 8 KB" case) was lower than pure read operations and slightly higher than pure write operations (see Table 4.5). Although the NPI interface provides doubled transfer rate in case of duplex operations than in the case of simplex operation, the overall throughput is influenced (probably) by lower performance of DDR2 interface in the duplex mode. The data-flow that was disturbed by short random bursts (the "8 KB, 16 KB" case) achieved 619 MB/s and 613 MB/s in read and write directions respectively. As the MPMC arbitrates the operation at the user interfaces in terms of bursts, the "disturbing" flow achieved low data throughput due to short length, and thus low utilization, of its bursts.

4.5.2 MPMC Core in Spartan6

The second set of experiments evaluates the MPMC throughput through software simulation. It is similar to the first set of experiments, but a different target FPGA is considered. The MPMC soft core was generated using the Xilinx XPS 12.4 tool. Transfer generators for the NPI interface were implemented through custom simulation models. A real DDR3 memory was simulated through a model of DDR3 memory by Xilinx.

The configuration of the MPMC core was as follows:

- MPMC Core version 6.02.a
- 2x NPI 64b @ 133 MHz
- Round-robin arbitration among user interfaces
- DDR3 16b @ 333 MHz

Table 4.6 Throughput of a single NPI port for read and write operations of different length; Spartan6

	Read [MB/s]			Write [MB/s]		
Transfer length	NPI#0	NPI#1	DDR3	NPI#0	NPI#1	DDR3
1 KB	522	508	1,030	516	538	1,024
4 KB	519	515	1,034	530	538	1,068
16 KB	518	517	1,035	536	538	1,074

4.5.2.1 Simplex Operations at a Single NPI Port, XC6S

The first experiment measured a throughput of a single NPI port. The goal was to evaluate the maximal throughput of either data transfer type (read or write) in the case when it is the only operation that is performed by the MPMC core at a time.

The ratio of the throughput of the NPI interface to the maximal theoretical throughput of the same interface was evaluated as well. The maximal theoretical throughput of a 64b interface running at 133 MHz is 1,066.7 MB/s.

An experimental workload considered in this experiment is the same as workload that was used in the experiment described in Sect. 4.5.1.1.

Results are shown in Table 4.6. The table shows NPI interface throughput in MB/s for read and write operations (columns 2 and 4, respectively) and ratio of these values to the maximal theoretical throughput (columns 3 and 5).

The experiment showed that the continuous data-flow through the DDR3 interface that was fed by two NPI interfaces was 1034 MB/s in case of read operations and 1074 MB/s in case of write operations. It is 50% and 65% of the highest throughput of read and write operations, respectively, at the Virtex5.

4.6 Technology Characterization

4.6.1 Analysis of Scaling Limits

The architecture was ported to several generations of Xilinx FPGA technology: Virtex 5, 6, and Spartan 6. In this paper we consider mainly the frequency characteristics of the technologies.

The BCE core is divided in two clock domains: (1) The MCU and the configuration interfaces are clocked at the base frequency f_0. (2) The VPU is clocked at f_{VPU}. Generally $f_0 \leq f_{VPU}$. Architecturally this clock domain bisection is quite optimal. As the MCU is part of a greater system with which it has to communicate (the γ, δ links in Fig. 4.3), and also because the PicoBlaze processor that implements the MCU operates in lower frequency ranges, it is clocked at the system base frequency f_0. Contrary, the VPU is coupled only through the command/status link α (that

carries the vector instruction word to the VPU) and indirectly over the dual-ported local memories (which implicitly separate clock domains without any additional provisions). Therefore, only the α link needs to be implemented with cross-domain flip-flop registers.

The base frequency f_0 is determined by external factors, such as the System-on-Chip platform and the Host CPU (e.g., MicroBlaze). For simplicity we assume here $f_0(\text{Spartan6}) = 50\,\text{MHz}$ and $f_0(\text{Virtex5} + 6) = 100\,\text{MHz}$.

The maximal f_{VPU} is determined by the level of pipelining in data paths. The separation of the DFU and AG units by FIFO queues allows to pipeline the two parts of VPU independently. For example, when the high-speed mode is configured, additional registers are inserted between the crossbar switch and the memory banks (increasing the effective latency of a memory read from 1 cycle to 3 cycles) as it was found that the Xilinx BlockRAMs can have relatively large delay on their output wires (more than 1.5 ns in Virtex 5).

The level of pipelining in DFU is determined by the latency of the floating-point arithmetic compute cores that are used. Ideally, increasing pipeline latency ($k_\circ$) of a unit by factor S should also increase the maximal operational frequency by the same factor. Thus, the absolute pipeline lead-in delay time when executing simple element-wise mapping functions stays constant, while the steady-state execution time (when the pipeline is full) is decreased by factor S. It is therefore always advantageous to deep-pipeline these operations.

However, windup delay w of full reductions increases superlinearly to the pipeline latency $k_\circ$ (Eq. 4.5). As the windup delay is only a constant part of the total execution time of a full reduction ($n + w$, Eq. 4.7), the total speedup depends on whether a given application executes the reductions with shorter or longer vectors.

Execution time (in seconds) of full reductions after scaling both the frequency and the pipeline latency by the factor S is given in Eq. 4.13:

$$T_{\Psi_\circ}(f, n) = \frac{1}{f} \cdot (n + (1 + \log_2 k_\circ)k_\circ - 1) \tag{4.12}$$

$$T_{\Psi_\circ}(Sf, n) = \frac{1}{Sf} \cdot (n + (1 + \log_2 (Sk_\circ))Sk_\circ - 1) \tag{4.13}$$

The goal of the scaling is to reduce the total execution time (Eq. 4.14). From that the minimal vector length n_{min} for which the speedup is achieved in the full-reduction operations is derived in Eq. 4.15:

$$T_{\Psi_\circ}(f, n) \geq T_{\Psi_\circ}(Sf, n) \tag{4.14}$$

$$n_{min} = 1 + k \cdot \frac{S \cdot \log_2 S}{S - 1} \tag{4.15}$$

Table 4.7 Tabulated values of Eq. 4.15

	S (scaling factor)			
$k_\circ$	1.5	1.66	2	2.5
2	5	5	5	6
3	7	7	7	8

Table 4.8 Maximal frequency in MHz depends on the pipeline depth. Ax = FP-Adder latency, Mx = FP-Multiplier latency

VIRTEX 5	M2	M3
A3	100M	
A4		150M
A5		166M
A6		

(a) Virtex 5

VIRTEX 6	M2	M3
A3	100M	
A4		150M
A5		
A6		200M

(b) Virtex 6

SPARTAN 6	M3	M4
A2	50M	
A3		<125M
A4		125M

(c) Spartan 6

The last equation is tabulated in Table 4.7. The pipeline latency $k_\circ$ is to be given before scaling both the frequency and latency by the factor S. The data values in the table represent the minimal vector length so that the reduction operation is *not slower* (in absolute time) after the scaling.

4.6.2 Synthesis Experimental Results

To determine the technology scaling characteristics the BCE core was instantiated in FPGA in a given configuration, and the maximal target operating clock frequency f_{VPU} was iteratively lowered until the FPGA synthesis process (including place, route, and timing analysis) finally succeeded. Table 4.8 summarizes the results for Virtex 5, 6, and Spartan 6 technologies. The BCE configuration is expressed in $AxMy$ notation: Ax specifies that the FP-Adder has latency x, and similarly My refers to the FP-multiplier latency y. The maximal operating frequencies of the VPU are 166 MHz, 200 MHz, and 125 MHz, for V5, V6, and S6 technologies, respectively.

The area breakdown in flip-flops (FF), combinatorial resources (LUTs), and slices is given in Fig. 4.13. The presented measurement was performed in the Virtex 6 technology; in the other technologies (V5, S6) the results are similar. The single-clock domain (low-speed) and dual-clock domain (high-speed) configurations are compared in the figure.

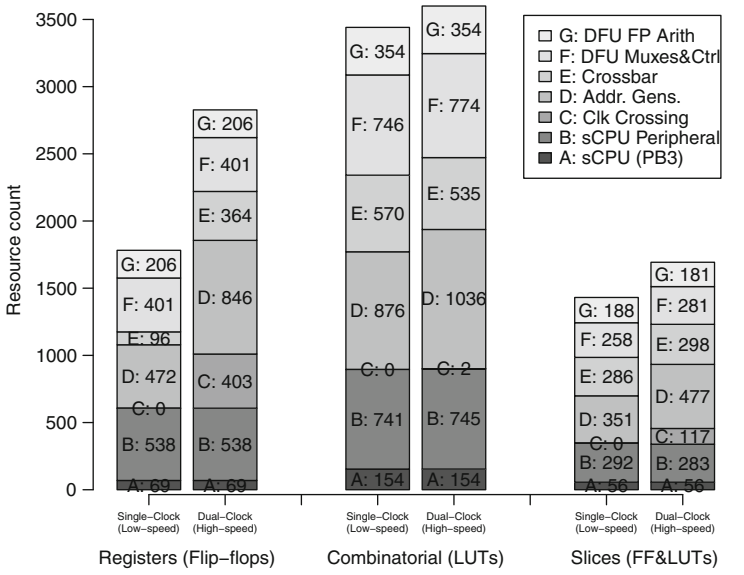

Fig. 4.13 Breakdown of the area (FF, LUTs, and slices) consumed by the BCE core in Virtex 6 technology in the single-clock/low-speed and dual-clock/high-speed configurations

The control processor (MCU, PicoBlaze 3) consumes very little resource itself. However, its peripheral circuitry (I/O decoders and multiplexers) consumes about 5× more space than the MCU. The registers for clock domain crossing between f_0 and f_{VPU} constitute about 15% of flip-flops in the dual-clock configuration. The relatively high count is caused by registering the whole vector instruction word α between the MCU and VPU. The dual-clock configuration is also optimized for higher frequency, thus additional registers are used in the VPU for deeper pipelining. The crossbar is about 3.8× larger in flip-flops because registers are instantiated on all its inputs. Thus the effective bank memory access latency is increased from 1 cycle to 3 cycles for data has to traverse the crossbar twice. (In the low-speed configuration the crossbar matrix is combinatorial.) Address Generators consume about 1.8× more registers because several more FIFOs are instantiated and the existing ones are deeper. The FIFOs serve two purposes in the architecture: First, FIFO queues equalize delay between pipeline head and tail; when pipeline's sink is blocked, the head stops issuing new data, but the pipeline itself is allowed to continue processing the data because the queue in the tail is guaranteed to store it. (A global pipeline stall signal can have potentially high fan-out that hinders routing performance.) And second, queues shorten combinatorial paths.

All in all, the dual-clock high-speed configuration requires 1.6× more registers but roughly equal number of LUTs. In Virtex 6 technology one slice represents 4 six-input LUTs and 8 storage elements. Thus, after packing the design into slices, the dual-clock high-speed configuration requires roughly 1.2× more slices.

4.7 Applications

4.7.1 Methodology

Given the technology performance characteristics determined by FPGA implementation in Table 4.8, we simulated the cycle-accurate synthesizable VHDL model of the BCE core in the ModelSim simulator to measure the execution time and dynamic profile of various benchmark kernels (FIR, MATMUL, MANDEL, IMGSEG—see below). Bar plots in Figs. 4.14–4.17 give the total execution time in microseconds for different technology nodes (latency and frequency) and kernel parameters (e.g., matrix size). The total execution time is split into four parts in the bars: (1) execution on MCU that was not overlapped with computation in DFU; (2) DFU has full pipeline (useful computation); (3) DFU pipeline bubble due to reduction windup; (4) DFU stall (input data not available).

The bar plots are overlaid with line plots that show the ideal clock frequency scaling: The first column in each group is taken as the baseline, and the subsequent data points are simply scaled by frequency difference. This ideal scaling is highly optimistic because it assumes that the whole BCE's frequency is increased, while in fact due to the implementation constraints we keep the MCU clock frequency f_0 constant and change only f_{VPU}.

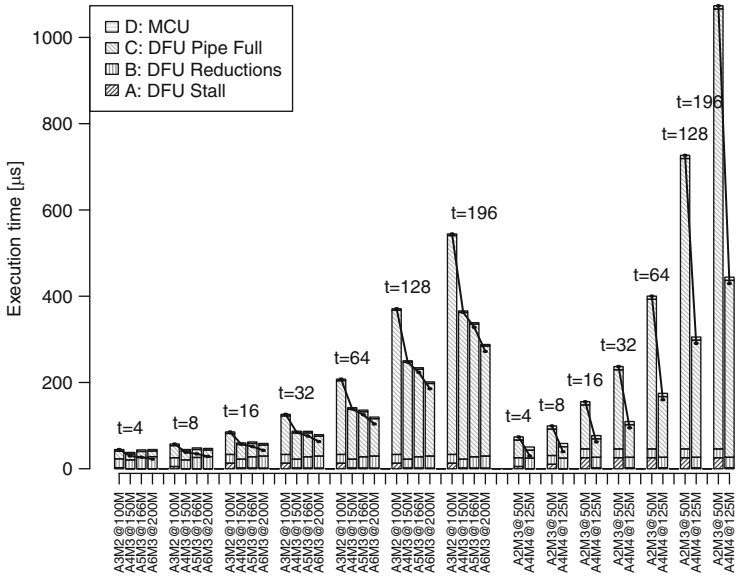

Fig. 4.14 *FIR Filter* execution time in microseconds for different number of filter taps t, technology (*left*: Virtex 5/6, *right*: Spartan 6), and the corresponding frequency/latency ratios

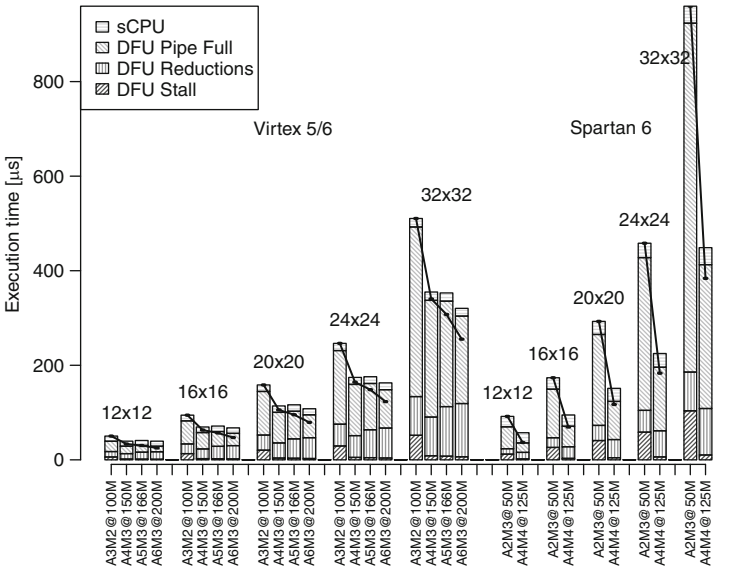

Fig. 4.15 *Matrix multiplication* (MATMUL) execution time in microseconds for different matrix sizes, technology (*left*: Virtex 5/6, *right*: Spartan 6), and the corresponding frequency/latency ratios

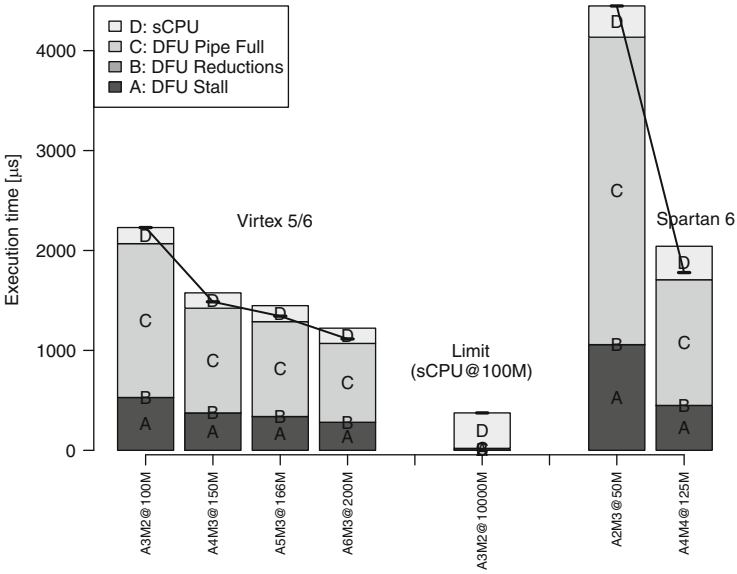

Fig. 4.16 *Mandelbrot set* (MANDEL) execution time in microseconds for different technologies (V5/6, S6) and the corresponding frequency/latency ratios. The *middle column* (labeled "Limit") shows the speedup limit imposed by the MCU

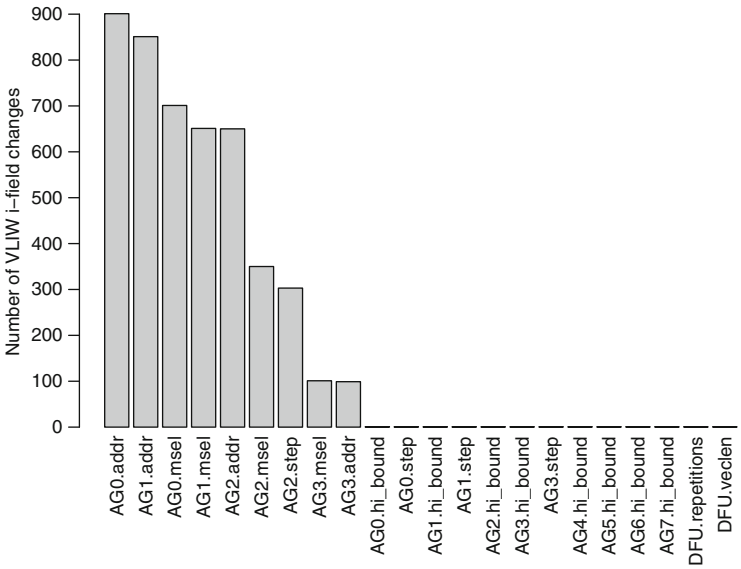

Fig. 4.17 *Image segmentation* (IMGSEG) execution time in microseconds for different technologies (V5/6, S6) and the corresponding frequency/latency ratios. The *middle column* (labeled "Limit") shows the speedup limit imposed by the MCU

All the benchmark programs compute in the floating-point single procession format.

In the more complex benchmarks (MANDEL, IMGSEG) the minimal kernel execution time caused by the fixed execution speed (f_0) of the MCU is presented. The time is measured in simulation by setting f_{VPU} to a very high value (10,000 MHz), hence fully exposing the MCU latency.

4.7.2 Finite Impulse Response (FIR) Filter

The FIR filter output is defined by the equation:

$$z_k = \sum_{i=0}^{t-1} (x_{k+i} \cdot b_i), \tag{4.16}$$

where z is an output vector (with n elements), x is an input vector (with $n + t - 1$ elements), b is a vector of coefficients (taps), and t is the length of vector b. We assume $n \geq t$.

The filter can be implemented using only one DFU vector instruction that computes n dot-products of length t each. Execution speed vs. the technology and latency is plotted in Fig. 4.14. Output vector length is constant for all measurements: $n = 255$. The FIR filter's execution speed scales very well because (1) The control overhead in MCU is constant for all tap lengths t as only one DFU instruction is required. (2) The reduction cycle count w is independent to the number of filter taps t.

4.7.3 Matrix Multiplication (MATMUL)

The MATMUL program (Fig. 4.7) computes dense matrix multiplication $C = A \times B$. For brevity we present only results measured on square matrices n x n. The computation requires n^2 dot-products that are grouped in n batches. Each batch computes one row (or one column) of the result matrix.

Matrix multiplication shows relatively high reduction cycle count overhead (Fig. 4.15). In the fastest Virtex 6 node A6M3@200M when multiplying 32×32 matrices the reduction cycles contribute about 35% of the total execution time.

4.7.4 Mandelbrot Set (MANDEL)

The MANDEL program computes the Mandelbrot set for a given set of points (tile). The output of the kernel is an array of the number of iterations executed for each point until the point is known not to lie in the set. The tile size is 200 points, and the maximal number of iterations before forced escape is 50. The kernel speculatively executes all 50 iterations for each point, hence its execution time is independent to the actual part of the set being computed. The body of the iterative loop consists of 18 vector instructions.

The fractal computation kernel (Fig. 4.16) does not contain reduction operations, and given its use of relatively long vectors (tile size 200 elements), it scales quite well because even in higher f_{VPU} the MCU has enough time to prepare another vector instruction in the forming buffer. Figure 4.18 shows histogram of MCU writes into op-code fields of the vector instruction forming buffer.

4.7.5 Image Segmentation (IMGSEG)

The IMGSEG program implements image foreground/background pixel motion detection (segmentation) using the algorithm presented in [9] without shadow detection and with several modifications. The decision if pixel from the current frame represents foreground or background depends on statistical models and their

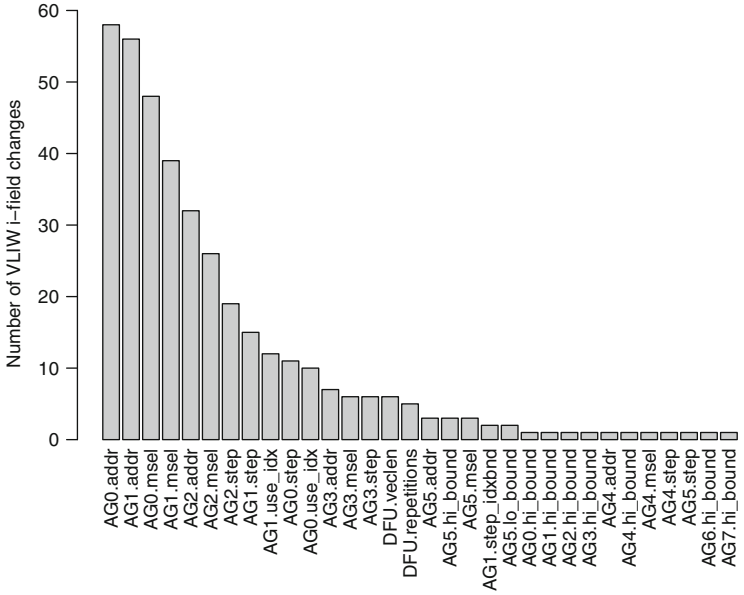

Fig. 4.18 *Mandelbrot set* (MANDEL): Histogram of alterations performed by MCU to the fields in the vector instruction forming buffer between two succeeding instructions

mixture. Each pixel is modeled by a mixture of K strongest Gaussian models of background (K=4 in our implementation); Gaussian models (mean values and dispersion) represent a state (color) of the pixel. Each model also contains the *weight* parameter that specifies how often the particular model successfully described background in the pixel. The algorithm was vectorized and the vector length is 50 pixels. Conditional branches were vectorized by speculatively computing both possibilities and then *VSELECT*ing the correct value per each element.

Execution time of the Image Segmentation kernel (Fig. 4.17) is largely determined by the MCU speed. In Spartan 6 A4M4@125M technology the VPU is idle 52% of the total execution time due to the MCU incapability to prepare the next vector instruction for execution before the previous one has completed. Source of the problem is hinted in Fig. 4.19, which presents the histogram of MCU writes into the instruction forming buffer. Comparing the histogram to the *MANDEL*'s one in Fig. 4.18 we see that the IMGSEG kernel writes much more operation fields in the instruction buffer during the execution.

The second source of prolonged execution time, albeit much smaller, is the reduction windup latency caused by many INDEXMAX/INDEXMIN instructions executed on very short (4-element) vectors.

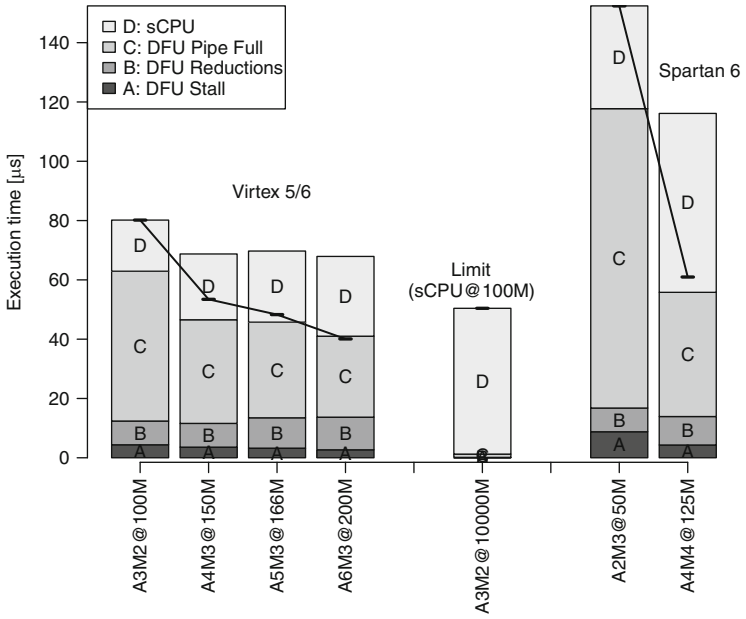

Fig. 4.19 *Image segmentation* (IMGSEG): Histogram of alterations performed by MCU to the fields in the vector instruction forming buffer between two succeeding instructions

4.8 Analysis of Weaknesses

4.8.1 Operation Frequency

It is not simple to determine the maximal operational frequency of the BCE (Table 4.8) because the core shall be used as a component of a larger system in an FPGA that places additional stress on the placing and routing process.

Specifically, we found that it is not possible to vary the f_{VPU} clock frequency independently of the f_0 system clock because some propagation delay is needed between rising edges of different clock domains. Timing closure is more easily achieved when $n \in \mathbb{N} : f_{VPU} = n \cdot \frac{f_0}{2}$.

4.8.2 FPGA Area

We intuitively expected that the crossbar will be the most resource hungry component of the BCE. Instead, Address Generators collectively occupy the largest share of resources. The situation can be improved either by function subsetting as not all features are used in every application (e.g., the lower/upper bound address detection)

or by reducing the replication of AGs (eight in Fig. 4.5). For example, the IMGSEG kernel always uses only one indexing AG at a time; thus AGs 4–7 could be merged into one shared unit. The other advantage of reducing the number of AGs is in lowering the required number of ports to the crossbar.

Flip-flop resource count needed for clock domain crossing could be improved if the domain boundary is moved closer to the MCU, i.e., the MCU I/O port demultiplexing, and the instruction forming buffer (Fig. 4.8) would be clocked in the f_{VPU} domain. This modification would narrow the required number of wires that must be resynchronized between clock domains effectively to the MCU I/O port interface.

Finally, we note that relatively large area is devoted to the MCU peripheral circuitry that essentially performs only demultiplexing of the I/O address space. We speculate that by using a 16-bit MCU this area could be reduced as less demultiplexing will be required; however, it is not immediately clear if the gain would be offset by the larger MCU.

4.8.3 Full-Reduction Windup Latencies

All the presented kernels (except MANDEL) contain some form of the full-reduction operators. For example, in the MATMUL kernel, the windup latency accounts for 35% of the total execution time when computing on 32×32 matrices, using Virtex6 A6M3@200M. The elementary vector operation in the MATMUL is a dot-product (DPROD). Dot-product computation of vector length $n = 32$, using pipelined FP-Adder with latency $k_+ = 6$, will experience windup latency $w = 21$ cycles (Eq. 4.5), and it will take $t = 53$ cycles to complete. However, of the 21 windup cycles, only 5 cycles will the FP-Adder input a new pair of values. As the dot-products in MATMUL (and similarly INDEXMAX/INDEXMIN in IMGSEG) are programmed to execute in batches, the DFU could be modified to interleave the windup execution in one operation with computation in the following one. The technique is similar to loop unrolling in scalar architectures.

4.8.4 MCU Performance

MCU performance was found to be a critical factor for IMGSEG kernel. We implemented the MCU using the PicoBlaze 3 (KCPSM3) processor, which executes one 8-bit instruction in two cycles. In Spartan 6 the MCU is running at 50 MHz; its performance is only 25MIPS. Contrary, the VPU's peak performance (in A4M4@125M) is 250MFLOPs, and it can output 125M-elements/s. IMGSEG executes 58 vector operations that collectively take 2,790 cycles of DFU time (counted in the f_0 domain), thus there are on average only 48 cycles of MCU execution that are fully overlapped between two vector operations. In 48 cycles

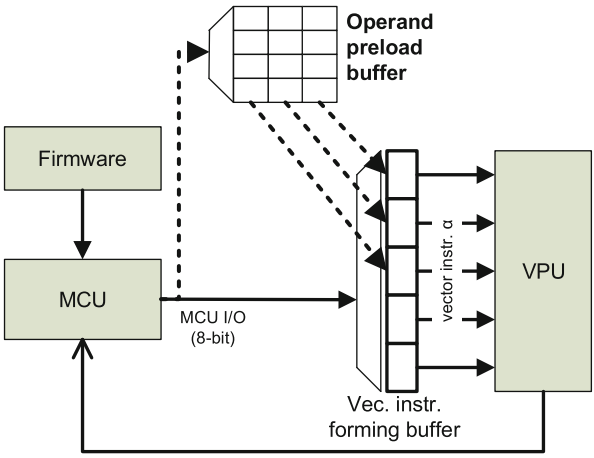

Fig. 4.20 The primary AG fields ("addr" and "msel") are quickly loaded from a new config. table

PicoBlaze 3 executes only 24 instructions. To load a single 16-bit value into the vector-instruction-forming buffer (e.g., AG initial addresses are usually changed between two succeeding operations), PicoBlaze processor has to execute four instructions (2xLOAD, 2xOUTPUT). Thus, the many instructions needed in MCU to prepare the next vec. instruction cannot be overlapped by useful computation in VPU. There are several possible approaches to the problem:

1. *Increasing MCU speed* by lifting one or several of the following issues: (a) PicoBlaze executes only one instruction in two cycles; (b) PicoBlaze's maximal operational frequency is limited by its internal long combinatorial paths; (c) narrow 8-bit data paths and instructions.
2. *Sharing a single DFU among several MCUs to improve efficiency* (exploiting task-level parallelism).
3. *Operand preload buffer:* Histogram plots in Figs. 4.18 and 4.19 show the frequency of alterations of op-code fields between two succeeding vector instructions. The "AG*n*.addr" and "AG*n*.msel" fields are very often altered from one vec. instruction to another for they define the starting address and bank of a vector variable in local memory. Hence we propose a new configuration table to store the memory locations of vector variables (Fig. 4.20). Each row in the table would correspond to one vector variable used in the application; the content of the table will be the initial address (10–16 bits) and the memory bank selection (2 bits). A row from the table could be loaded into the proper fields in the vec. instruction forming buffer by a single MCU port write that specifies both the table row (port I/O data, 8-bit) and the Address Generator (by a part of the I/O address). The proposed modification would lower the PicoBlaze instruction count required to reload the primary AG fields ("addr" and "msel") from 6 to 2.

4.9 Summary

The new version of the EdkDSP platform represents a major leap forward in our understanding of the Universe. We improved both the user-visible application programming interface (API) and the internal hardware organization. Innovations in the user-visible API were guided by the needs of the Image Segmentation application. This includes new addressing modes, new operations, and provisions for DMA-streamed memory accesses.

Internal organization of the platform was largely overhauled. In the previous version data processing was centralized in a single unit (DFU) that handled all addressing, processing, and control. In the new version we have recognized the orthogonality of address generators, crossbar, and data processing unit. This in turn allowed us to easily abstract DFU operations as combinations of a control loop with an imbedded static data-flow circuit.

References

1. H. P. Hofstee, "Heterogeneous Multi-core Processors: The Cell Broadband Engine," in *Multi-core Processors and Systems*, ser. Integrated Circuits and Systems, S. W. Keckler, K. Olukotun, and H. P. Hofstee, Eds. Springer US, 2009, pp. 271–295. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-0263-4_9
2. C. Kozyrakis and D. Patterson, "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 35. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 283–293. [Online]. Available: http://dl.acm.org/citation.cfm?id=774861.774892
3. C. Kozyrakis and D. Patterson, "Overcoming the limitations of conventional vector processors," in *Proceedings of the 30th annual international symposium on Computer architecture*, ser. ISCA '03. New York, NY, USA: ACM, 2003, pp. 399–409. [Online]. Available: http://doi.acm.org/10.1145/859618.859664
4. R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The Vector-Thread Architecture," in *Proceedings of the 31st annual international symposium on Computer architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 52–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006736
5. P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: portable, scalable, and flexible FPGA-based vector processors," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, ser. CASES '08. New York, NY, USA: ACM, 2008, pp. 61–70. [Online]. Available: http://doi.acm.org/10.1145/1450095.1450107
6. J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, "Vector Processing as a Soft Processor Accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 12:1–12:34, June 2009. [Online]. Available: http://doi.acm.org/10.1145/1534916.1534922
7. J. Kathiara and M. Leeser, "An Autonomous Vector/Scalar Floating Point Coprocessor for FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, may 2011, pp. 33 –36.

8. S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 31. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 3–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=290940.290946
9. P. Kaewtrakulpong and R. Bowden, "An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection," 2001.

Part III
Run-Time and Faults Management

Chapter 5
Fault Tolerance

Giovanni Agosta, Mickael Cartron, and Antonio Miele

5.1 Introduction

The current trends in technology, fabrication processes, and computing architectures are increasingly pushing towards the design and development of multi-core and many-core systems constituted by a relevant number of relatively low-cost execution resources (e.g., processors and configurable accelerator units) to achieve high performance while leveraging on energy consumption. These trends must cope with increasingly unreliable devices, affected by the shrinking of component size, variations in the manufacturing process, and increased transient errors caused by radiations and noise fluctuations.

In particular, the *variability* phenomenon increases and causes speed variations of about 30%, due to several mechanisms:

- First, the number of doping atoms in the transistors' channels is decreasing exponentially with the technology evolution. Considering the random fluctuations of these doping atoms, the threshold voltage is not constant. For recent technologies, the mean number of doping atoms is about several tenths, and small fluctuations cause detectable variations.
- Second, the UV lithography has reached a limit due to the wavelength since 65 nm, causing an increasing variability ever since.
- Third, as the power per surface unit increases, some hot spots appear, creating significant temperature variations.

G. Agosta (✉) • A. Miele
Politecnico di Milano, Piazza L. da Vinci 32, I-20133, Milano, Italy
e-mail: agosta@elet.polimi.it; antonio.miele@polimi.it

M. Cartron
CEA, LIST, Laboratoire de Fiabilisation des Systèmes Embarqués, Point Courrier 94, Gif-sur-Yvette, F-91191, France
e-mail: mickael.cartron@cea.fr

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_5,
© Springer Science+Business Media New York 2014

The second reliability issue is the *aging*, which slows the transistors down along with their usage and/or with the time. The aging is due to the reduction of the saturation current in the channel, which is due to several mechanisms:

- The *electromigration* (EM) is a transport of atoms. This effect increases with the current density, which means that it is more important than ever with recent technologies.
- The gate oxide can be subject to wear out due to various reasons. Among them, the *hot carrier injection* (HCI) is due to fast electrons or holes that integrate into the oxide layer and affect the threshold voltage of transistors. The gate oxide is more sensitive to wear out in recent technologies.
- Lastly, the *negative bias temperature instability* (NBTI) is a phenomenon that modifies the threshold voltage of PMOS transistors under high temperature conditions.

Then, recent technologies are increasingly sensitive to *soft errors* [single error transient (SET) and single error upsets (SEU)]. These phenomena are mostly due to particle strikes (alpha particles, neutrons, protons), which cause electric charge modifications in the circuits. When a critical charge Q_{crit} is exceeded, the logic state may be modified. This Q_{crit} parameter reduces with the technology scaling, thus increasing the soft error probability.

The SMECY project focuses on many/multi-core systems and the size of this kind of systems is by itself a reliability issue, increasing the variability and the soft error occurrences. Besides this, the multiprocessor system can have multiple applications and/or threads, and errors can propagate from one to another if nothing is done to prevent it.

Achieving a better fault tolerance is a problem that can be treated at several levels. Resources in many-core systems tend to be uniform and interchangeable. Also, all available resources are often not used at 100%, and the unused resources may be used for fault-tolerance purpose. In order to minimize processes' failures, two kinds of approaches may be used: *proactive* or *reactive*. The *reactive approaches* consist in combining three compulsory mechanisms:

- A *containment* mechanism (fault or error containment)
- A *detection* mechanism (fault or error detection)
- A *recovery* mechanism

Given the many-core context of the project, various combinations of these different mechanisms were explored, at a rather high level:

- New detection mechanisms are proposed at task level (Sects. 5.4.1, 5.4.2 and 5.4.4).
- Fault containment is also addressed (Sects. 5.4.6 and 5.4.3).
- For permanent faults, the recovery strategies mostly consist in resource relocation (processor, memory) and a solution to make this relocation easier is proposed in

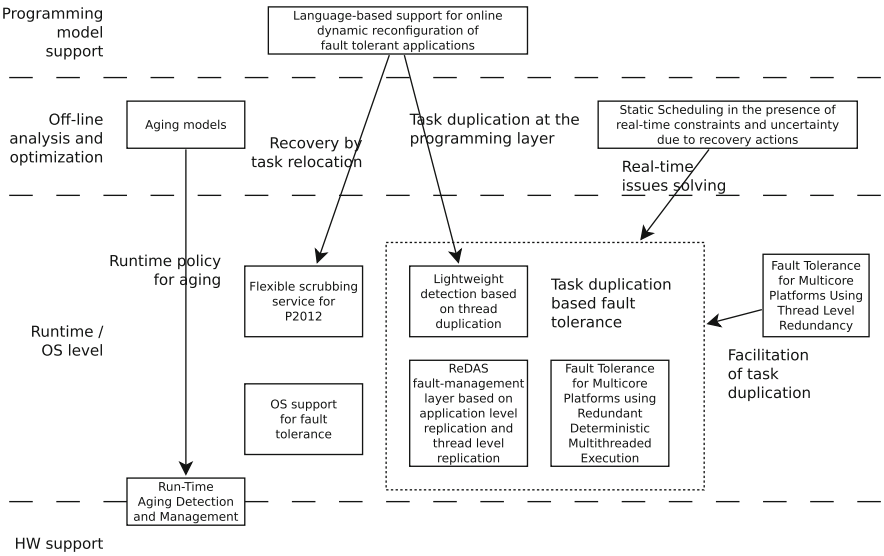

Fig. 5.1 Global diagram of online fault-tolerance solutions proposed for the SMECY project

Sect. 5.2.1. Also, recovery strategies cause some uncertainties in the execution flow, which is, a priori, not suitable for real-time application. A solution to this problem is proposed in Sect. 5.3.1.

Besides the reactive approach, the *proactive mechanisms* were also addressed. Indeed, with many-core systems, we have to deal with nonuniform aging in the whole chip due to a combination of factors: the technology scaling, which tends to increase variability problems, and the nonuniform usage pattern, which tends to alter the circuits at different speeds. Solutions to these problems are proposed in Sects. 5.4.5 and 5.4.4. The idea is to propose a model for the system aging and to use this data online to perform proactive wear out leveling.

A schematic overview of the different contributions to the SMECY project is given in Fig. 5.1. The proposed approach is organized in four layers, covering different levels of abstraction at which the fault-tolerance problem can be addressed. In the SMECY project, the software layers have been addressed, and the most relevant approaches are briefly discussed in the rest of the chapter.

5.2 Programming-Model Support Level

In this section, we introduce language-based support mechanisms for online dynamic reconfiguration of fault-tolerant applications.

5.2.1 *Language-Based Support for Online Dynamic Reconfiguration of Fault-Tolerant Applications*

Dynamic resource allocation is an important feature for reliability in a many-core context. For proactive reliability, as already mentioned before, the wear out leveling can be used against aging. This consists in periodically modifying the resources in use in order to get a uniform aging and enhance the lifetime of the system. For reactive reliability, dynamic resource allocation is also an important feature which is necessary for dealing with permanent faults. When a resource has caused errors several times in similar conditions, a mechanism at run-time is necessary for moving the threads to healthy resources.

This task relocation problem was studied in the SMECY project. The consortium explored the usage of the concurrent programming model of *occam-pi* [1], combining communicating sequential processes (CSP) [2] with *pi-calculus* [3] to model the dynamic reconfigurability available in the hardware and to exploit this for fault-tolerance features. The programmer should be able to easily describe the computations and to match them to the target hardware. Occam-pi is a programming model that allows an explicit expression of concurrency, as well as the notion of memory spaces. It is also possible to express dynamic parallelism, dynamic process invocation mechanisms. This language is generic enough to be adapted to a wide class of embedded computing architectures.

This programming model was adapted to the STHORM platform, under joint development by STMicroelectronics and CEA.

The front end of an existing translator from occam to C from Kent (Tock) was reused and a new back end for targeting the STHORM platform was developed. The front end of the compiler consists of phases up to machine-independent optimization and the back end includes the remaining phases that are dependent upon the target machine architecture. The front end of the Tock compiler consists of several modules which perform operations like lexical analysis, parsing, and semantic analysis. The STHORM back end targets the whole platform and its integration with the host system as shown in Fig. 5.2.

The STHORM back end generates the complete structure of application components in native programming model (NPM) and a host-side program to deploy, control, and run the application components on the STHORM fabric. An NPM application is designed by using:

- The architecture description language (ADL), which describes the structure of each component
- The interface description language (IDL), which describes the components' interfaces
- An extended C code, which is an implementation of the code that runs on the basic processor element of a STHORM cluster

The generated code can then be executed in the GePOP simulation environment, provided by STMicroelectronics. After the application is designed, to deploy,

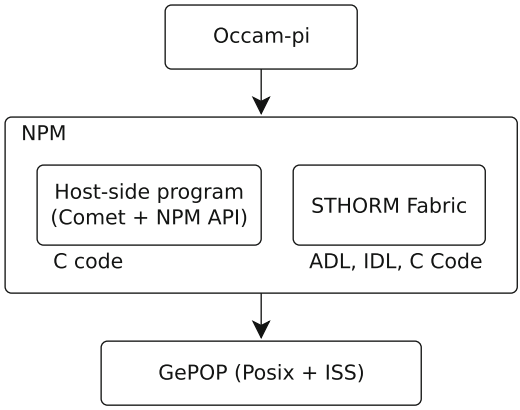

Fig. 5.2 Compilation of Occam-pi to STHORM platform

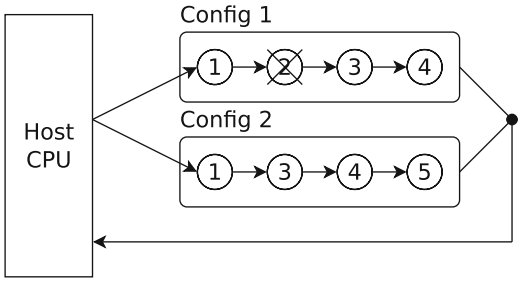

Fig. 5.3 Demonstration of dynamic reconfiguration of STHORM architecture

manage, and run the application, a host-side program must also be developed. For this, NPM and the Comete tool, which is a middleware for loading, deploying, and managing software components, are used. More details on the translation from occam-pi to NPM can be found in [4].

An application case study was implemented making use of the constructs available in the occam-pi language to realize a fault-tolerance strategy on the STHORM platform. The application illustrates the use of occam-pi to manage the relocation of tasks from faulty to healthy processing elements on the P2012 fabric. The application consists of 1D-DCT algorithm which is implemented on four encore processing elements of Cluster 0 of P2012 fabric as shown in Fig. 5.3.

In the first configuration the DCT algorithm is mapped to four processing elements marked 1–4. During the execution of the DCT kernel, the run-time system detects error in one of the processing element and passes the error code to the application, which communicates the error-message together with their corresponding processing element ID to the host processor. The host processor after receiving an error-message will issue a new configuration deployment by making use of FORK construct of occam-pi, where the error code is passed as parameter to the forked process. This new forked process will result in generation of the next configuration where the identification information about the faulty processing element is used to avoid its use in the deployment of next configuration by the run-time system.

5.3 Off-Line Analysis and Optimization Level

In this section, we investigate static approaches to mapping and scheduling multi-threaded applications on many-core architectures, taking into account the fault-tolerance aspects.

5.3.1 Static Scheduling in the Presence of Real-Time Constraints and Uncertainty due to Recovery Actions

Fault-tolerance mechanisms such as task duplication and the corresponding recovery actions (e.g., re-execution) ensure the correct application behavior but have adverse effect on the system performance and on the ability to satisfy real-time constraints. This is particularly relevant for non-preemptive scheduling, widely used on clustered domain-specific data processors under the supervision of a general-purpose CPU such as the STHORM platform.

In such a scenario, real-time guarantees can be provided via static off-line scheduling, provided the execution delay due to potential faults is properly handled. A typical approach consists in modeling delays due to recovery actions as a source of uncertainty on task durations; then some robustness can be added to an off-line computed schedule by inserting slack time (i.e., idleness periods) between start times of communicating tasks [5] to account for possible re-executions; at run-time, one is not allowed to anticipate any tasks, since any deviation from the off-line schedule may result in so-called anomalies [6], eventually leading to uncontrolled performance degradation and the possible violation of real-time constraints. The obvious drawback is a considerable loss of efficiency, due to the inserted idle time.

The goal is to perform predictable and efficient non-preemptive scheduling of multitask applications in the presence of uncertainty. The target platform is assumed to be described in an abstract fashion as a set of resources with finite capacity. An application is a set of dependent tasks, modeled as a graph where execution/communication time is characterized by known min–max intervals and unknown probability distribution. The primary focus is on the scheduling procedure (i.e., ordering tasks over time): the resource mapping is assumed to be available via some external tool. In particular, the proposed approach leverages a hybrid off-line/online technique from the Project Scheduling domain in Operations Research and Constraint Programming; the specific technique is known as precedence constraint posting (PCP) [7].

PCP is a hybrid off-line/online scheduling technique. The main idea is to compute off-line a partial schedule, consisting of a set of additional arcs in the task graph; at run-time, the schedule is implemented by delaying each task until all its predecessors (both natural and artificial) are over. As a consequence, the completion time adapts to actual task durations and unnecessary idleness is avoided. Note that a partial schedule corresponds in practice to a large set of possible actual schedules, depending on which fault effectively occurs.

The additional precedence constraints are carefully designed so as to avoid any possible resource conflict at run-time, effectively eliminating the risk of anomalies. As a consequence, the partial schedules have predictable worst-case behavior and can provide real-time guarantees, provided the maximum delay on each due to recovery action is bounded.

Possible conflicts are identified as minimal critical sets (MCSs), i.e., minimal sets of tasks causing a resource overusage in case of overlapping execution. Thanks to the minimality property, an MCS is wiped out (resolved) by adding a single precedence constraint between a pair of involved tasks; an MCS-free graph is anomaly-free as well. Finding MCSs is a nontrivial task, since their number is exponential in the size of the graph. The set of precedence constraints to be added can be obtained via tree search [8] or by means of a heuristic.

5.3.1.1 Formal Problem Definition

The problem addressed is that of scheduling a set of interdependent tasks with uncertain duration over a set of preassigned hardware resources, in the presence of hard real-time constraints. Formally, the input for the optimization process is a directed, acyclic Task Graph $G =< T, A >$, where T is the set of tasks t_i and A is the set of graph arcs (t_i, t_j), representing precedence relations. Each task t_i is annotated with a minimum and a maximum duration bounds; the bounds account for the maximum delay possibly incurred due to faults and recovery actions, but they may be used as well to model duration variability due to data-dependent computation. Each arc (a_i, a_j) is labeled with a minimum and a maximum time lag (referred to as $\delta_{i,j}$ and $\Delta_{i,j}$), so that the time distance between the end of t_i and the beginning of t_j is forced to be larger than $\delta_{i,j}$ and smaller than $\Delta_{i,j}$.

In detail, a minimal time lag is a deterministic separation constraint. A maximal time lag is a relative deadline, i.e., the maximum allowed distance between a pair of dependent tasks. It is possible to use minimal time lags to represent latency due to inter-task communications, but in many cases it is more reasonable to model non-instantaneous communications as fake tasks.

A release time (or earliest start time) and a deadline (or latest end time) can be specified for each task and are denoted as $\text{est}(t_i)$ and $\text{let}(t_i)$. The execution of t_i must take place within the interval $[\text{est}(t_i), \text{let}(\text{t}_\text{i})]$ and the time lags must be respected, whatever the actual activity durations are. A set of platform resources R is part of the input; each resource r_k has limited capacity c_k and each task t_i requires an amount $rq_{i,k} \geq 0$ of every resource r_k ($rq_{i,k} = 0$ denotes no requirement). Those resources can be used to model computation cores, storage devices with limited capacity, or a bus or a memory port with limited bandwidth.

Figure 5.4 shows an example of a problem instance. Note that duration variability due to faults is captured by using bounds (e.g., extracted via WCET analysis relying on some fault model) rather than via a probability distribution, to reduce the computational complexity. As discussed in the next sections, this is sufficient to guarantee the satisfaction of hard real-time constraints; as a drawback, bounds

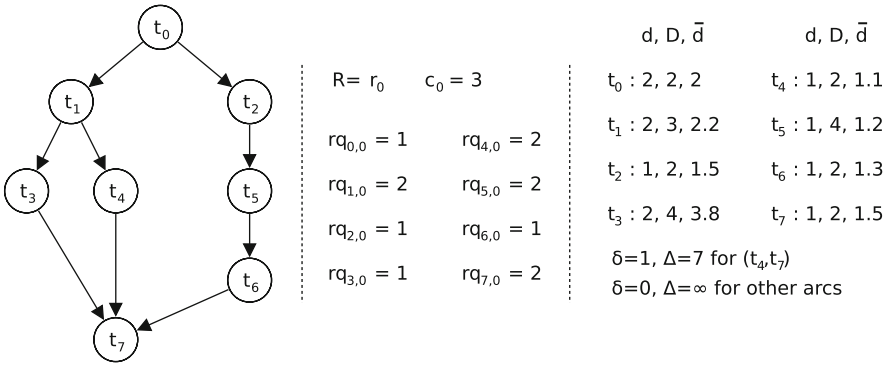

Fig. 5.4 A sample problem instance (i.e., description of a target application and platform)

give poor information on what the application completion time will be in a typical case: to cope with this issue, each task is labeled with its average duration value d_i. This is used to compute a rough approximation of the average completion time. In detail, the off-line problem consists in computing a schedule such that temporal and resource constraints are satisfied for every possible value of task durations; furthermore, the optimizer tries to minimize the completion time corresponding to the case where all tasks assume their average duration. Note that arc (t_4, t_7) in the example is annotated with some time lags.

5.3.1.2 Scheduling Problem Solution

Classical off-line scheduling methods for this scenario assume worst-case task durations (i.e., D_i) and specify a fixed starting time of each task. Figure 5.5a shows an optimal fixed start schedule for the example problem; due to time lags, t_7 must be scheduled at least one time unit after t_4 and t_4 cannot be scheduled more than seven time units earlier than t_7. In case actual durations are shorter than D_i, idleness is inserted to preserve the start times, typically resulting in a strong efficiency loss.

Ideally, one would like to be able to anticipate activities in case their predecessors last shorter. This goal can be achieved by adopting a flexible scheduling approach, i.e., by computing an off-line partial schedule, where some decisions still have to be taken. In particular, the schedule is specified as a set $\square$ of additional synchronization constraints (i.e., precedence relations). Those may be implemented with little run-time overhead as fake data communications or by maintaining counters for the number of completed predecessors. Figure 5.5c shows an example of such a partial schedule, consisting in the added arc (t_4, t_5); note the augmented graph corresponds to a huge set of possible schedule instantiations, depending on the actual duration values. Figure 5.5b finally shows the corresponding completion time in case all task assume their average duration (i.e., the quality measure optimized by the off-line scheduler).

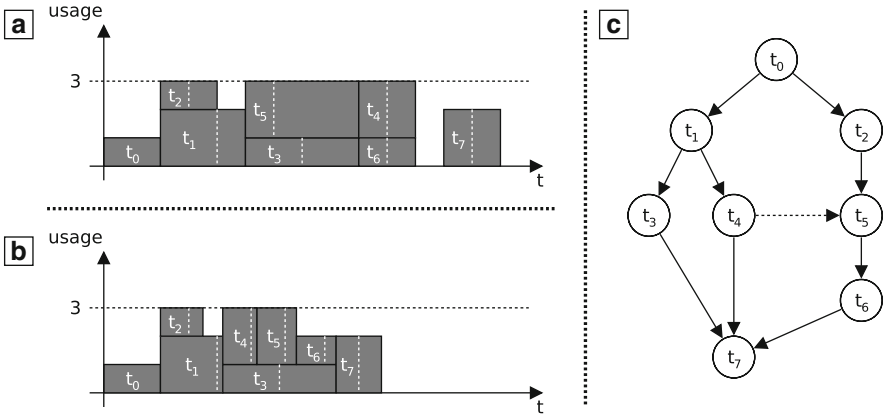

Fig. 5.5 (**a**) Fixed time start schedule; (**b**) completion time for the PCP schedule in (c) when all tasks have average duration; (**c**) PCP schedule

A partial schedule is completed at run-time by starting each task t_i according to an earliest start policy, namely, as soon as (1) t_i has been released and (2) all its predecessors (both actual and artificial) have completed execution. The additional precedence relations are carefully computed so as to prevent resource contention. No resource conflicts mean no scheduling anomalies, so that a partial schedule has predictable worst-case behavior. In the partial schedule in Fig. 5.5c, arc (t_4, t_5) has been added to prevent the resource conflict caused by the overlapping execution of t_4 and t_5.

5.3.1.3 Solution Method

The described problem can be solved by means of Constraint Programming and tree search; each branching node represents a temporally feasible, resource infeasible partial schedule. The search proceeds by checking the presence of a MCS and testing all the precedence constraints that can be used to resolve it. Search stops when the augmented graph contains no MCS, so that the capacity of every resource is not exceeded in any online generated schedule. The basic method to check the presence of an MCS consists in analyzing all possible combinations of independent tasks: unfortunately, this approach has exponential complexity. Instead, MCS is detected in polynomial time by coupling a flow minimization problem and a greedy set minimization procedure. State-of-the-art heuristic techniques are used to decide the evaluation order of the MCS to be considered for branching and of the precedence constraints used to resolve them.

At every search node, the feasibility of temporal restrictions for every possible combination of duration values is ensured by modeling uncertain durations and temporal constraints by means of a simple temporal network with uncertainty (STNU;

see [9]). An STNU has nodes, representing temporal events, and arcs, constraining the time distance between the occurrence of the source and the target event to be in an interval [min, max]; in particular, STNUs distinguish between free constraints (the distance can be decided at execution time) and contingent constraints (the time distance can only be observed). An STNU is said to be dynamically controllable if, during execution, the distance for free constraints can be decided so that the partial sequence executed so far is ensured to extend to a complete solution, whatever durations remain to be observed. Dynamic controllability is enforced by means of a more flexible method based on a constraint model; more information on the temporal model and on the solution method in general can be found in [10].

Our tree search procedure is designed to find a partial schedule guaranteed to satisfy all temporal and resource constraints; the approach is complete, i.e., it always returns a feasible graph augmentation in case this exists. However, the basic procedure does not perform any minimization; therefore, optimized schedules are obtained by progressively tightening a threshold on the maximum allowed average completion time (approximated as mentioned in the previous section), according to a binary search scheme.

The approach has been tested on synthetic benchmarks, designed to represent realistic workloads. A comparison of the PCP-based method with a pure online first in first out (FIFO) scheduling approach and Priority-Based Scheduling (optimized off-line via tabu search) was carried out; despite the fact that these methods cannot provide real-time guarantees, they are widely used in practice to schedule for low completion time in a dynamic environment (such as a system subject to faults). The results are very promising: the PCP schedules exhibit good stability and even improved average completion time.

5.4 Runtime/OS Level

In this section, we discuss several run-time task duplication techniques for fault detection and tolerance explored in the SMECY project in order to take profit of resources that may remain unused.

5.4.1 Lightweight Detection Based on Thread Duplication

An error detection method based on thread and communication buffer duplication was implemented for the SMECY project. The duplicated resources are the processors (the two threads' instances do not run on the same processor), the private memory, and the streaming buffers between threads. The streaming buffers that may be used are duplicated so that they can accept two different writers. The checking is performed by an analysis of the streaming buffers. All features are integrated in a simple POSIX compliant API in a run-time software for platform STHORM.

5.4.1.1 Description of the Approach

The proposed technique is a fault-detection algorithm at task level, with low performance overhead. It exploits the fact that some resources of a many-core cluster may be unaffected or at least not fully used. The basic principle is to use the unaffected resources and exploit them as redundant resources to enhance dependability. The redundancy is managed at software level which allows to easily select where the dependability service is applied. This also allows to adapt the method to other cluster architectures.

The proposed technique allows the detection of errors in the outputs of a thread, due to faults in various hardware resources, such as memory, bus, or processors. It allows to check the outputs of threads, but this verification is done by a procedure which can be scheduled in various ways, depending on the global application. A high fault containment at the level of one task can be achieved by checking the output every time a data is produced. But it is also possible to provide the fault containment at application level, when the application is made of several tasks. This allows reducing the cost dedicated to the detection. This choice has to be made at application level. This is the originality of this work, contrary to Redundant Multi-Threading, which is a lower-level technique that exploits SMT hardware in processors. The proposed method makes it possible to implement a less tightly coupled fault-containment strategy in order to optimize the performance overhead and cost of the reliability approach at system level. When errors are detected, the recovery consists in calling a service in the run-time software that performs an application restart. This recovery frees the resources affected to the application that is affected by an error before freeing the memory and cleaning the task information in the kernel. A schematic of the modular run-time designed by CEA LIST is represented on Fig. 5.6. In this figure, only POSIX API parts being linked with the duplication method used are represented.

The scheduling policy of the thread management system is FIFO based and cooperative. The concept of *sphere of replication* (SoR) (inspired from Mukherjee [11]) can be defined as a logical domain of redundant execution. Outside of this SoR, the logical domain is seen as a unique black box where the redundancy is hidden. The SoR is responsible for the management of the duplication of inputs and outputs. Some inputs of a SoR need to be replicated to allow the twin mechanisms to progress at their own pace. There are as many output flows as the number of replicated instances and the outputs are compared for fault detection or fault masking (voting). One queue is considered as the main output queue and the other is considered as a shadow queue. Depending on the application, the output comparison may be done either at each production step or in a more sporadic way. The cost of the comparison depends on the content of the output stream and for that reason, it is very variable. If this cost is too high, a systematic comparison may not be acceptable.

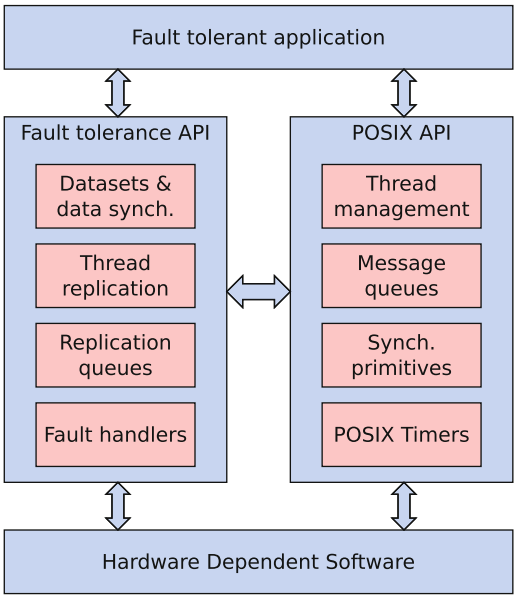

Fig. 5.6 Fault-tolerance API organization

5.4.1.2 Implementation Issues

The STHORM platform's cluster is operated with a run-time software developed by CEA LIST and STMicroelectronics. Briefly stated, our case study is a cluster composed of 16 processor cores with shared memory. Here, cache memories are deactivated. The STHORM run-time software provides a native programming layer with POSIX-like APIs. More details are given in [12].

The thread duplication method described before was implemented by the way of a replication library which was added to the native thread creation API. An additional argument *data_set* gives a description of data items that are accessed by the replicated threads. This new version can create a main thread and N background copies. The execution engine is based on a cooperative scheduler. The executing engine always tries to schedule the thread shadow on a processor different from the one allocated to the main thread, for increased fault coverage. The run-time can internally distinguish between a thread and its copies by assigning a special thread identifier (id).

The comparison action between both instances is available through the task duplication API, which allows to implement various fault containment strategies. The most strict fault-containment is achieved by doing the comparison every time both threads have written a new data in the output queue. This strategy may cause performance loss due to the output resynchronization. What is more, the comparison itself has a cost due to a software parsing of both queues. If we consider that the SoR is authorized to contaminate the next thread of the streaming chain—which means

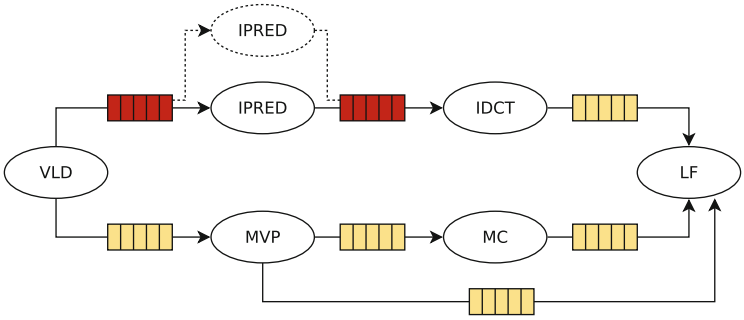

Fig. 5.7 VC1 task graph with partial replication

then that we assume that a recovery mechanism is available at a higher level for the management of this error propagation—then a less intensive scheduling of the comparison should be used. This decision depends on the application and on the availability of the recovery service.

5.4.1.3 Assessment

The performance, memory size, and fault coverage of the replication API were evaluated with the VC1 decoder application. The task graph of VC1 is shown in Fig. 5.7. It is composed of six threads; each thread is either a producer or a consumer. Inter-thread communications are performed through FIFO queues. Here, only thread *IPRED* is duplicated (dashed line) on a separate processor. An analysis of the application susceptibility to SEUs with, e.g., fault injection campaigns would aid to select the right parts to be replicated. Both input and output FIFO queues of this thread use the replication API (in red). There is no use of datasets in this example. The two thread copies share the same inputs. The comparison is performed at each write operation in the output FIFO queue. The extra execution time of the whole application is equal to 12% with an input/output bitstream of 64/75 kbytes (two frames). The overhead is mainly due to the extra bus accesses from the shadow thread, the waiting time for synchronization between the two threads (blocking FIFO write) and the data comparison. The main contribution is the latter one.

The comparison time increases linearly with the size of the buffer to be checked. The comparison is performed on 32-bit words. The memory size of the replicated application is increased by 2.5%. The overhead is mainly due to the additional code. The exhaustive assessment of fault coverage with the aid of fault injection techniques is hard since the exploration space of possible fault scenarios becomes combinatorial. SEU occurrences in processor and interconnect and shared memory (thread inputs and outputs) are taken into account. For the purpose of this

experiment, the effect of a bit-flip is simulated by forcing manually the content of variables of *IPRED* thread. With the nonreplicated version, it is observed that the effect of a bit-flip is either masked or benign (transient noises in output image) or severe (false coloring, black sequence). With the replicated version, any bit-flip effect that corrupts the outputs of one of the two threads is detected. Conversely, an SEU in thread inputs (common mode) or in a processor control register is not detected and may cause a crash.

5.4.2 Flexible Scrubbing Service for P2012

Software methods such as memory scrubbing can be employed against transient and permanent memory errors. When applied at operating system level, these methods have some drawbacks: the balancing between error coverage and performance cost is difficult and these techniques are reserved for read-only memory sections such as code sections. The flexible scrubbing service for P2012 is controlled at application level for scheduling the scrubbing with the knowledge of the application scheduling. This allows to maximize the error coverage while minimizing the performance cost. In addition, it enables to do memory scrubbing on read/write memory sections as well as read-only sections.

5.4.3 OS Support for Fault Tolerance

Current operating system originates from research in the 1960s and 1970s when the focus was to efficiently share expensive resources, such as a single processor, between users and programs. Over many years, the systems have been optimized and can scale well for traditional server-based, high-performance workloads. However, platforms, requirements, and workloads are changing. Platforms are now parallel with abundant compute resources. Energy and power have become precious resources which have to be conserved. Reliability and security requirements are becoming increasingly strict for many applications. Finally, workloads are changing, becoming more diverse and less predictable. As a consequence, innovation in operating systems is needed to address the changes in the computing landscape.

FenixOS is an operating system that aims to improve the state of the art in scalability and reliability. To achieve scalability, it limits the data sharing between processing cores. When data sharing cannot be avoided, the OS uses lock-free data structures. To achieve reliability, the programming interface and structure of the operating system have been carefully redesigned.

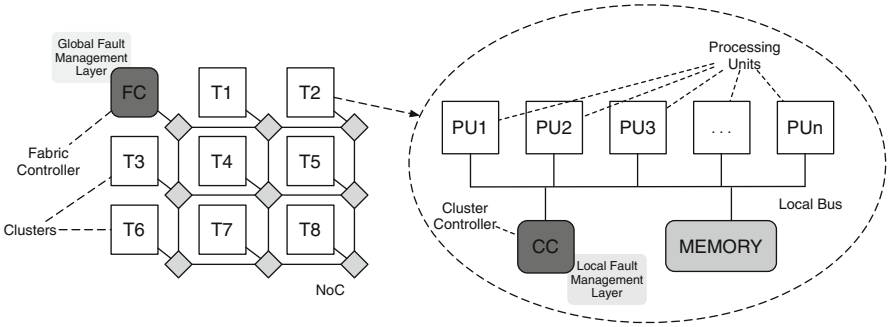

Fig. 5.8 The considered many-core platform provided with the two fault-management layers

5.4.4 ReDAS, Fault-Management Layer Based on Thread Level Replication

ReDAS is a *Reliability-Driven Adaptive Scheduling* approach for many-/multi-core architectures, aimed at providing a fault-tolerant execution of multithreaded applications. The ReDAS approach is based on the redundant execution of the application threads on the available healthy processing units, where redundant execution is used not only to achieve fault tolerance but also to monitor the status of the architecture resources, and their usage, in pursuit of an extended lifetime of the platform.

The approach has been designed taking in account a target many-core platform, shown in Fig. 5.8, composed by a set of *clusters* interconnected by a network-on-chip (NoC) and coordinated by a unit called *fabric controller*; each cluster is a multiprocessor system managed by a *cluster controller* and devoted to the fast execution of multithreaded applications; an example of such architecture is the ST/CEA P2012 [2]. The proposed approach is to dynamically adapt the exploitation of the system resources and the execution of multithreaded applications to cope with the occurrence of both soft and hard faults, avoiding those units identified as damaged. The main goal is to maximize performance, while balancing the use of the healthy resources to limit wear-out and aging effects, leading to permanently damaged units.

ReDAS consists of two different layers introduced at scheduling level in the considered architecture (refer to Fig. 5.8): at the fabric level, where the application is dispatched to the available clusters, as well as locally, in the cluster, where the processors and hardware accelerators are exploited to execute the application threads.

The *local fault-management layer* has been introduced in the cluster. In particular, the cluster controller is in charge of guaranteeing the correctness of the computations during the execution of the single application sent by the fabric controller. A few assumptions have been adopted. Due to this central and critical

role, the cluster controller and the hardware synchronizer are assumed to be designed as hardened at the architectural level. This is necessary to avoid that a fault affecting their activity would not be detected and would cause the failure of the overall cluster execution without any alert (single point of failure). Moreover, the cluster has a memory protection mechanism allowing each thread to write only its private variables, input data, and output data. This protection system prevents a thread affected by a fault from randomly corrupting the memory of the other threads running on the other cluster's processing units, leading to the failure of several threads. Finally, the existing watchdog timers are exploited for detecting violations of execution deadlines caused by transient/permanent faults. Given these assumptions, any physical permanent or transient fault can be modeled as a failure of the single processing unit running a thread producing a wrong result within the expected deadline.

The cluster fault-management layer adopts a detection and mitigation strategy based on replicating the execution of the application two or more times, on different processing units, to compare redundant results; in particular, triple modular redundancy (TMR) is used for mitigating errors. When a new application is received, the cluster controller issues three different replicas of its execution. Then, the cluster controller's scheduler that uses a FIFO policy has been enhanced to assign replicas of the same nominal thread on different processing units. Moreover, after completing the execution of a replica triplet, the scheduler issues a voter task devoted to the detection and the mitigation of errors generated by the occurred faults. Due to the criticality of the voting threads, the cluster controller executes them, it being the only fault-tolerant unit. Finally, successor threads become ready only after the end of the voting activities.

The fault-management layer is also in charge of classifying faults as transient/permanent and of adopting the appropriate recovery actions. For this purpose, the layer uses two support tables:

- The *local resource health table* (LHT) stores information on the status of cluster and its processing units. In particular, each processing unit may assume three different states: (1) healthy, if the unit is operational, (2) damaged, if a permanent failure has been identified, and (3) under-analysis, if the unit is suspected to be damaged and, therefore, requires the execution of specific diagnostic procedures.
- The *error log table* stores information on the errors occurred in the computation and identified by the voting threads, to determine whether the fault is transient or permanent (on the basis of its occurrence).

Periodically, the cluster controller performs an analysis of the error frequency in each processing unit. If the frequency of a processing unit reaches a specified threshold, the layer can put that resource in quarantine by updating in the resource health table its status to under-analysis. In this way, the processing unit becomes unavailable for the scheduler, and a diagnosis thread is started on that resource to verify the presence of a permanent damage. If the test is positive, the unit is tagged has damage and will not be used; otherwise, the resource status is restored to healthy

and the unit will be again available to the scheduler. Finally, the fault-management layer is provided also with additional scheduling policies to prevent and limit wear-out and aging effects. In particular, LHT contains another column specifying the utilization of each processing unit. Thus, to balance the average workload of each unit, when there are new ready threads, the scheduler will awake the processing units according to the order specified by this new index.

A second fault-management layer has been introduced in the fabric controller to perform a reliability-aware dispatching of the applications to be executed. The fabric controller has a global health table (GHT) that summarizes the status of the overall architecture containing information about the percentage of damaged resources and a utilization index of each cluster. Similarly to the state-of-the-art P2012 implementation, the entire application is dispatched to a single cluster. Thus, a basic dispatching strategy aimed at limiting wear-out and aging effects has been defined, by allocating the application to be executed to the first idle cluster, with the minimum utilization index. Moreover, since the performance achievable by a cluster (and hence by the entire architecture) is affected by the ratio of its healthy resources, the fabric controller keeps track of the health status of each cluster in the GHT and when the number of healthy resources diminishes below a fixed threshold, it disables the cluster as a whole.

5.4.5 Run-Time Aging Detection and Management

Among the various dimensions of reliability, aging (i.e., a drift of some performance metric over time) has emerged as a critical problem in nanoscale technologies, due to the manifestation of mechanisms such as bias temperature instability (BTI), HCI, and time-dependent dielectric breakdown (TDDB). These effects directly impact the aging of devices (logic and memories) with the net result, on a larger scale, of progressively decreasing the operating frequency of a core. This problem, which is relevant by itself, is even more marked in multi-core architectures, where, due to nonuniform usage patterns, the various cores will age at different rates. Letting a device completely wear out without any intervention will impact the system as defective cores have to be permanently removed from the pool of active cores. Strategies include (1) detecting the aging, possibly online and without resorting to sensor, and (2) combating the nonuniform aging over the various cores.

Here, we focus on a methodology for the construction of high-level aging macromodels to be used dynamically to drive strategies that aim at equalizing the aging of the various cores. The methodology consists of two phases: model design and model characterization. There is a possible third phase (model adaptation) that depends on the availability of (or the possibility of inserting) aging sensors on the target platform.

5.4.5.1 Model Design

Since the goal is to have macromodels that depend on quantities that can be extracted during platform execution/simulation, the first step is to identify the model parameters. Parameters that satisfy this requirements are typical execution-related metrics such as those provided in typical performance counters (e.g., L1/L2 misses, idle cycles, pipeline stalls). The selection should be the most general possible to avoid too core-specific parameters.

Once parameters are selected, the model is defined. Here there are several options. First one is to have a flat model in which all execution-related parameters and the environmental ones (temperature, supply voltage, etc.) are in the same equation. Another option is to have a two-level model in which the execution-related parameters are used to model a sort of *activity factor*, and a classical analytical model is used for aging, modulated by the activity factor.

More precisely, in the first case, the aging (e.g., Δf-frequency decrease) could be expressed as a general (linear) equation

$$\Delta f = F(execution\ parameters,\ physical\ parameters).$$

For instance, $F = a_1 N_{L1misses} + a_2 N_{idlecycles} + b_1 T + b_2 V_{dd}$ would be a concrete example.

In the second case, the model would be

$$f = F_1(execution\ parameters) \times F_2(physical\ parameters)$$

For example, $f = (a_1 N_{L1misses} + a_2 N_{idlecycles} + \ldots) \times F_2(V_{dd}, V_{th}, T, \ldots)$, where F_2 is the analytical formula describing the basic physics of the modeled phenomenon.

5.4.5.2 Model Characterization

Characterization of an aging model based on measures is difficult if not impossible because it would imply to execute the system for a large amount of time: to observe a meaningful amount of performance drift the system should be run (or simulated) for times in the order of weeks if not months. Therefore, empirical fitting of the models based on measurements is impossible.

Exploiting the property that most aging phenomena, and NBTI in particular, are value-dependent, and that this value dependence is independent of the temporal distribution of values and depends only on the occurrence probability of such values, we can resort to probabilistic simulation. By using probabilities the time horizon is irrelevant, so a single simulation can be used to represent a behavior of any temporal length.

A probabilistic simulation clearly requires the availability of the netlist of the core to actually simulate and extract signal probabilities. While this is not always possible (as, for instance, would be for the P2012 core), by properly selecting the parameters of the models (the execution parameters), and by using a generic core architecture, one can derive the model and assume it has a more or less general validity.

The selected netlist is then RTL simulated under various testbenches to extract several sample points. Every simulation will get a given value of the chosen parameters and the corresponding aging value. The various samples will then be used to derive the coefficients of the model using regression.

Once the model is determined, it can be easily used in the target platform (e.g., P2012) to extract aging. This will require the calculation of the various model parameters for a given time interval (e.g., miss rate and pipeline stalls over 10K cycles), plug them into the models, and get the resulting aging data.

5.4.6 Fault Tolerance for Multi-Core Platforms Using Redundant Deterministic Multithreaded Execution

A common technique applied for fault tolerance is redundant execution, in which a program is replicated in such a way that replicas process inputs in the same order, perform the same computations, and produce the same outputs. In other words, the replicas need to be deterministic.

In this way, if any divergence is found between the replicas, it can be assumed that it is due to a fault. However, the problem with multithreaded applications running parallel on multi-core systems is that they are inherently nondeterministic, especially if they are programmed in traditional programming languages such as C and use mutexes to perform shared memory accesses. Moreover, there are other sources of nondeterminism such as interrupts and nondeterministic functions, such as random numbers.

To address these issues, a library is proposed to allow programmer to execute replicas in a deterministic manner in the presence of mutexes and nondeterministic functions. Moreover, such a library allows the programmer to checkpoint the application at regular intervals, so that when divergence is seen due to a fault occurring in one of the replicas, the application is rollbacked and reexecuted from that checkpoint. Lastly, a watchdog timer can be used to initiate rollbacks on hang-ups. Figure 5.9 shows the resulting control flow for fault-tolerant execution.

At this moment, the proposed approach only addresses transient faults. However, deterministic execution can also be efficiently applied for fault diagnosis, that is to find if a fault is transient, permanent, a software bug, or a hardware fault. An initial implementation on general-purpose platforms demonstrates an overhead limited to less than 50%.

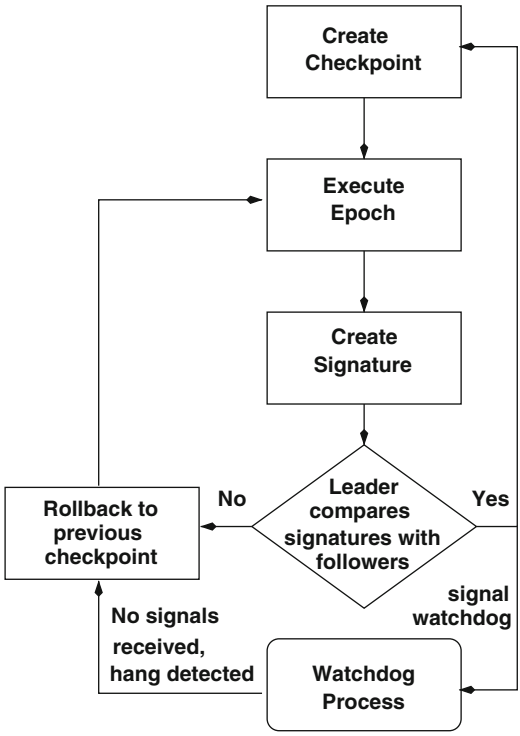

Fig. 5.9 Control flow for fault-tolerant execution

5.5 Conclusion

In the SMECY project, the issues related to the reliability of emerging many-core architectures to be used for the acceleration of compute-intensive applications have been addressed through several approaches, targeting different levels in the combined space of design and run-time tools and software stacks.

Techniques investigated include extensions to the programming model to support the introduction of fault-detection and fault-tolerance capabilities. Such mechanisms are then used by approaches working at design time or at run-time. A key challenge is the unpredictability of the working conditions (application workload) of the system. A many-core platform, given its abundance of resources, can be used through a fault-tolerance-aware run-time resource management system, to replicate critical computations to ensure the identification of faults and the mitigation of their effects. Another key challenge is the issue of aging, that is the degradation of hardware components with time. This issue has been tackled both at the software level and, to a lesser extent, at the hardware level, thus enabling a graceful degradation of the system.

In conclusion, it can be stated that the availability of resources in excess of those needed to guarantee performance can be effectively used to manage fault tolerance in software. At the same time, the homogeneity of resources allows the

ability to achieve a graceful degradation by applying run-time resource management techniques to isolate the failing components and redistribute the workload on the remaining processing elements.

References

1. P. H. Welch and F. R. M. Barnes, "Communicating mobile processes: introducing occam-pi," in *In 25 Years of CSP*. Springer Verlag, 2005, pp. 175–210.
2. C. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Ed. Prentice-Hall, 1985.
3. R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part i," 1989.
4. Z. ul Abdin, E. Gebrewahid, and B. Svensson, "Managing dynamic reconfiguration for fault-tolerance on a manycore architecture," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 312–319.
5. A. J. Davenport, C. Gefflot, and J. C. Beck, "Slack-based techniques for robust schedules," in *Proc of ECP*, 2001, pp. 7–18.
6. J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *Proc. of WCET*, 2006.
7. N. Policella, A. Cesta, A. Oddi, and S. F. Smith, "From precedence constraint posting to partial order schedules," *AI Communications*, vol. 20, no. 3, pp. 163–180, 2007.
8. P. Laborie, "Complete mcs-based search: Application to resource constrained project scheduling," in *International Joint Conference on Artificial Intelligence*, 2005.
9. T. Vidal, "Handling contingency in temporal constraint networks: from consistency to controllabilities," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 11, no. 1, pp. 23–45, 1999.
10. M. Lombardi, M. Milano, and L. Benini, "Robust scheduling of task graphs under execution time uncertainty," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 98–111, 2013.
11. S. Mukherjee, *Fault Detection via Redundant Execution*. Morgan Kaufmann, 2008, ch. 6, pp. 207–251.
12. STMicroelectronics and CEA, "Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology," 2010, whitepaper.

Chapter 6
Introduction to Dynamic Code Generation: An Experiment with Matrix Multiplication for the STHORM Platform

Damien Couroussé, Victor Lomüller, and Henri-Pierre Charles

6.1 Introduction

Since the early beginning of computer history, one has needed programming languages as an intermediate representation between algorithms description and machine-readable instructions. In broad outline, running an algorithm on a computer requires the following steps: (1—software development, implementation) the developer transcribes the algorithm into a source file containing programming language instructions, (2—compilation) a compiler translates these programming language instructions into machine code and performs adaptations to the original code for optimized fit to the target execution support, and (3—execution) the processor reads and executes the machine instructions, loads the input data and produces the data results.

Because compilation is performed *before* the program is run, the execution context and runtime data are not known at the time of code generation (Fig. 6.1a). In order to leverage such information in code optimizations, several solutions exist. The first is to assume about the characteristics of the execution context (and to provide verification mechanisms). Another solution consists in adding extra instructions to adapt the program behavior depending on runtime data, which is known as code specialization. The last solution, which is the topic of this chapter, is runtime code generation. The aim is to generate the program's machine code at runtime, after the execution context and the optimizing runtime data are known.

Dynamic code generation can be achieved by interpretation or compilation at runtime [1]. In classical frameworks, the aim is to provide a generic infrastructure for code generation, bounded by the syntactic and semantic definition of a programming language. The generality of such solutions comes at the expense of

D. Couroussé (✉) • V. Lomüller • H.-P. Charles
CEA, LIST, DACLE/LIALP, F-38054, Grenoble, France
e-mail: damien.courousse@cea.fr; victor.lomuller@cea.fr; henri-pierre.charles@cea.fr

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_6,
© Springer Science+Business Media New York 2014

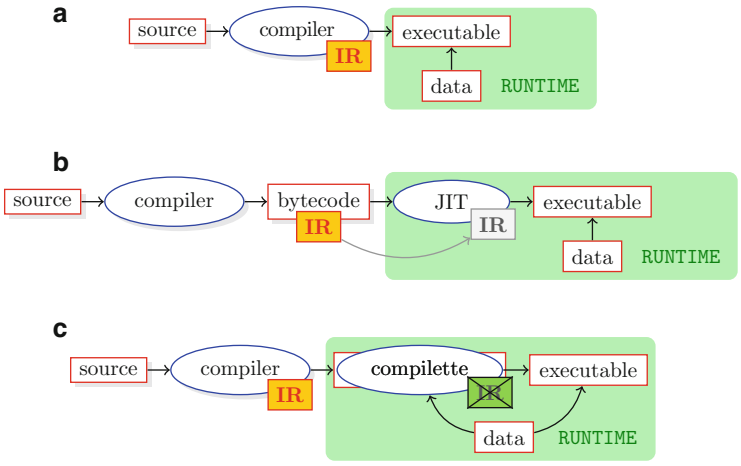

Fig. 6.1 Illustration of the static and dynamic compilation schemes and comparison with the runtime code generation with compilettes. *IR* stands for intermediate representation (**a**) static compilation (**b**) dynamic compilation (Java, etc.) (**c**) runtime code generation with deGoal compilettes

an important overhead in code generation, both in terms of memory footprint and computing power. A well-known example is the Java programming language, designed to enhance application portability: Java source code is written without a priori knowledge of the platform that will execute the final machine code, thanks to a virtual machine that relies on an intermediate representation, the Java bytecode (Fig. 6.1b). At runtime, the bytecode is either interpreted or compiled into machine code as soon as the overhead of code generation can be amortized by repeated calls of the generated code [2]. Despite the fact that a virtual machine has all required information to perform data-dependent optimizations, interesting values are difficult to use for such systems owing to an already high code generation cost [1].

Code optimization from runtime information is also useful for large-scale parallel computing systems, where an application component can be populated on a lot of processing elements. This application component has to be parametrizable so that its behavior can be adapted to the processing element where it is instantiated. To do so, one would need either (1) a generic implementation that one can parametrize at instantiation but that will suffer from the performance overhead brought by a generic implementation or (2) to modify and recompile the component dynamically at runtime after one knows where it will be finally executed. Being able to specialize the executed code for each of the computing elements is likely to provide performance improvements, as long as the cost for such optimization remains modest. This issue is applicable to all large-scale multiprocessor platforms: from high performance computers in data centers to multiprocessor Systems-on-Chip (MPSoCs) in future embedded devices. Due to the distributed nature of computing and memory resources in many-core platforms, it becomes challenging

to bring dynamic compilation capabilities to such platforms. Moreover, because of the non-negligible memory footprint of the runtime librairies of just-in-time compilers (JITs), the limited size of the local memory in embedded many-core platforms becomes another important bottleneck in this context.

deGoal was designed to provide application developers the ability to implement application kernels tunable at runtime depending on the execution context, on the characteristics on the target processor, and furthermore on the *data to process*: their characteristics and their values [3]. Usually in processing applications, most of the execution time is spent in a very small portion of the whole application source code, which is most of the time a computation-intensive task also called *kernel*. We assume that improving the performance of kernels can leverage the overall application performance. Therefore, the idea using deGoal is to embed *ad hoc* runtime code generators, called *compilettes*, in a software application. Each compilette is specialized to produce the machine code of one application kernel. On the contrary to dynamic compilation, in our solution we embed at runtime only the necessary processing intelligence to perform code optimizations that can exploit the properties of the data to process, but no analysis of the intermediate representation or a subset such as bytecode (Fig. 6.1c). As a consequence, this enables the production of very fast code generators (10 to 100 times faster than typical frameworks for runtime code interpretation or dynamic compilation). As such, deGoal provides a lightweight solution for dynamic code generation applicable to massively parallel systems. The compilettes offer a low-memory footprint and very fast code generation. Furthermore, deGoal was designed to provide very large portability, which makes it easily applicable to heterogeneous platforms: The compilettes are compiled from ANSI-C source code after source-to-source code transformations. The code generation process that we propose here can target a large number of platform architectures, which is only limited by the availability of a C compiler for the processor that will perform the code generation at runtime.

In this chapter, we present an approach to describe a specialized code generator. The aim is to build a system that:

- Minimizes the generation overhead compared to classical JIT systems
- Allows more flexibility over the generated function application domain Specifically, we want to be able to select the data type at runtime
- Brings gain in performance, or at least similar performances, by removing dead code, unused loads or by constant propagation

Our main contributions are:

- The presentation of a way to describe how a code generator should behave for a key part of an algorithm
- To illustrate that taking into account runtime environment for auto-tuning is possible and how it offers a performance improvement
- To illustrate the use of specialized code generation for the STHORM platform.

The rest of this paper is organized as follows: Sect. 6.2 introduces the core idea of deGoal and data-dependent code optimization and Sect. 6.3 details the use of

our tool on matrix multiplication for the processors of a MPSoC and the results achieved. We end this chapter by providing an overview of the related works in Sect. 6.4.

6.2 Overview of deGoal

6.2.1 Kernels and Compilettes

The two categories of software components around which our code generation technique is organized are called *kernels* and *compilettes*.

Kernel. A kernel is a small portion of code, which is part of a larger application, and which is the target of our runtime code generation setup. Our technique focuses on the optimization at runtime of these small parts of a larger application in order to improve the kernel's performance. In the context of the typical use of deGoal, good performance is understood as one or several criteria among low execution time, low-memory footprint and low energy consumption.

Compilette. A compilette is designed to generate the code of *one* kernel at runtime. A compilette can be understood as an *ad hoc* small code generator that is executed at application runtime. We use the term *compilette* to underline the fact that in order to achieve very fast code generation, this small runtime generator does not embed all the optimization techniques usually carried out by a static compiler, but only the required ones considering the target kernel to optimize.

In order to target computing architectures that include domain-specific accelerators and to raise the level of abstraction of the source code, compilettes are described using a mix of standard C and of a dedicated high-level ASM language: `Cdg` [3]. This language has demonstrated its ability to achieve performance improvements in comparison with highly optimized static code [4]. We have chosen to stay with an assembler-like language in order to stay as close as possible to the final runtime model: an instruction-set processor. Our aim is furthermore to allow the direct use of multimedia arithmetics and to provide flexible and easy support to vectors and complex data sets.

The main paradigm shift relies in the fact that `Cdg` instructions describe code to be generated instead of code to be executed. On the contrary to common ASM languages, it is possible here to parametrize these instructions with values known at runtime, and to use vector variables. The variables manipulated are vector registers, whose size will be determined at the time of code generation, when the use of the physical registers in the programming context is known. It is also possible to map the assembly instructions to vector instructions when they are available on the target processor, and to map the assembly instructions to different arithmetic operators depending on the data values to process. As we will illustrate in Sect. 6.2.3.2, it is possible to mix `C` instructions and `Cdg` instructions. In this case, the C source code will control the code generation done in the `Cdg` instructions.

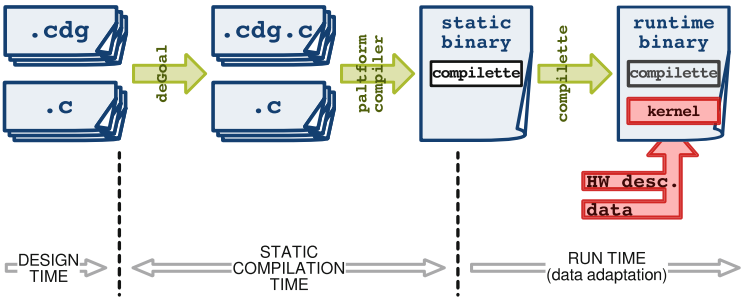

Fig. 6.2 deGoal workflow: from the writing of application's source code to the execution of a kernel generated at runtime

The instruction set includes:

A variable length register set. The instruction set uses vectorial registers with variable width and a variable number of elements, i.e., the programmer could define `Type f float 64 8` to use any register of type `f` as a vector of 8 elements of 64-bit floating point values.

Classical arithmetic instructions. `add`, `sub`, `mul`, `div`, but also instructions specific to the multimedia domain such as `sad` (sum of absolute differences), `mma` (matrix multiply and add) and FFT butterfly. These instructions can work on registers of variable length and type.

Load and store. This family of instructions supports stride description. This permits the description of complex memory access patterns.

Using this high-level instruction set, deGoal can generate the corresponding instructions for processors which have native support or generate optimized code for processors without support. In both cases the code generation is fast and produces efficient code.

6.2.2 *Workflow of Code Generation*

The building and the execution of an application using deGoal consists of the following steps: writing the source code; compiling the binary code of the application and the binary code of compilettes using static tools; generating the binary code of kernels by compilettes; running the kernels. These steps are illustrated in Fig. 6.2 and are explained below:

Application development time: writing the source code This task is handled by the application developer and/or by high-level tools. The source code of compilettes is written in specialized `.cdg` source files that allow for the mix of `Cdg` and `C` languages, while the rest of the application software components are written using a standard programming language, such as C.

Rewrite time: generation of C source files This step consists in a source-to-source transformation: the `.cdg` source files are translated into standard `C` source files by `degoaltoc`, which is one of deGoal tools.

Static compilation time: compilation of the application The source code of the application now consists in a set of standard C source files, including the source code of the compilettes. The binary code of the application is produced by a standard C compiler. This step is the same as in the development of a standard C application.

Runtime: generation of kernel's binary code At runtime, the compilette generates optimized binary code for the kernel(s) to optimize. This task can be executed on a processor that is different of the processor that will later run the kernel. Furthermore, the compilette's processor and the kernel's one do not necessarily need to have the same architecture. A compilette can be run several times, for example, as soon as the kernel needs to be regenerated for new data to process. We have detailed in Fig. 6.2 two particular inputs of the compilette: data and hardware description. The originality of our approach indeed relies in the generation of a binary code optimized for a particular set of application data. At the same time, the code generation is able to introduce hardware-specific features.

Runtime: kernel execution The program memory buffer filled by the compilette is run on the target processor (not shown in Fig. 6.2).

6.2.3 A Tutorial Example

Our tutorial example illustrates how to handle simple kernels for scalar multiplication using deGoal (Fig. 6.3). We introduce the main concepts of deGoal with the trivial example of the multiplication of two integer variables. We then elaborate on vector multiplication. For the purpose of illustrating how code generation is performed, our examples are based on the STxP70 processor, described in Sect. 6.3.2.1. However, the source code of the compilettes illustrated here could be applied straightforward to other processor architectures.

6.2.3.1 Simple Multiplication

We want to perform the runtime specialization of the generic function `genericMul` that multiplies two integers (Fig. 6.3a). After specialization, the function will be replaced by a function that multiplies by a constant known at runtime, i.e., that specializes the `val` parameter of `genericMul`. However, this parameter can only be known at runtime: at the initialization time of the process or during the program execution. Furthermore, it is likely to change multiple times.

The compilette `mulCompile` is a standard C function that includes elements of the `Cdg` language at lines 3 to 8 between `#[` and `]#` (Fig. 6.3b). Line 4 marks

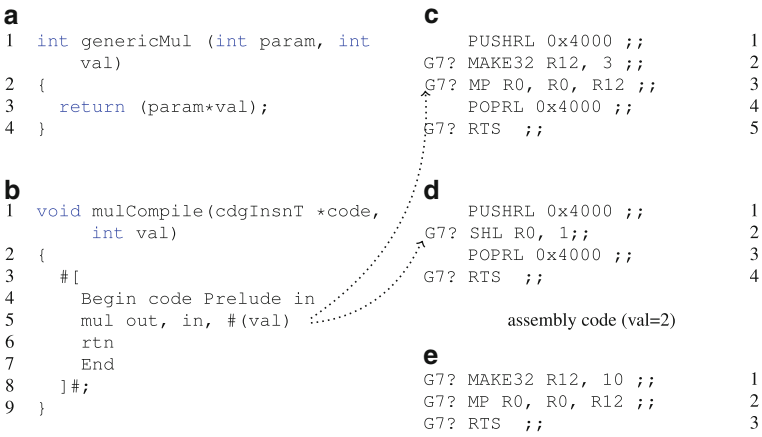

Fig. 6.3 A tutorial example: dynamic specialization of multiplication. (**a**) Generic code in `C` (**b**) compilette code (in `Cdg`) (**c**) assembly code (val = 10) (**d**) assembly code (val = 2) (**e**) assembly code (val = 10) for a leaf kernel

the moment where the code generation actually begins. `Prelude` states that this block needs stack and register management: in the generated code, we only save and restore the R14 register (link register) because R0 and R1 are defined as scratch registers in the ABI (application binary interface) of the STxP70. `code` is the pointer to the code cache, and finally `Prelude` comes with one parameter: `in`, which means that the generated kernel will take one parameter named `in`. According to the ABI of our target processor, `in` will be allocated by default on `R0`.

Finally, the `rtn` instruction is the return instruction that ends the kernel routine and inserts the return instruction. `End` ends the code generation: during code generation, the evaluation of this instruction triggers the computation of branch locations and the flushing of internal data.

Line 5 performs the multiplication between register `in` and a C r-value [written inside `#()`] and stores the result in `out`. In this case, the r-value is simply `val` `out` is the output register, allocated on `R0` for the STxP70. The compilette, when called at runtime, produces a binary kernel for the architecture selected at compilation time (Fig. 6.3c, d, respectively, when `val` equals to 10 and 2). The two dotted arrows highlight the locations where the runtime value `val` is evaluated and integrated into the produced code as a constant. In this tutorial example we illustrate a simple data-dependent optimization: the compilette generates either a kernel that uses the standard multiplication instruction (Fig. 6.3c) or the shift left instruction (Fig. 6.3d) depending on the value taken by `val` at runtime.

The source code of the compilette (Fig. 6.3b) is statically processed by deGoal. The specialized code generator is then generated and dumped into a C file that is statically compiled by the compiler of the target platform. This approach removes any direct intermediate representation manipulation and the need for complex code generation. This way, we reduce the computation time of code generation to the minimum.

```
1  void dot_product (int * A, int A_len, int alpha, int * B) {
2      for (int i=0; i<A_len; i++) {
3          B[i] = alpha * A[i];
4      }
5  }
```

Fig. 6.4 A trivial implementation of scalar multiplication for vectors in C

```
1  void compilette(cdgInsnT* code, int * A_addr, int A_len, int
       alpha) {
2  #[
3    Begin code Prelude B_addr
4    Type  int32       int 32
5    Type  vectorInt32 int 32 8
6    Alloc int32       tmp
7    Alloc vectorInt32 v
8
9    mv tmp, #(A_addr)
10   lw v, tmp
11   mul v, v, #(alpha)
12   sw B_addr, v
13   rtn
14
15   Free tmp, v
16   End
17 ]#;
18 }
```

Fig. 6.5 Implementation of a compilette for scalar multiplication with vector registers. For the sake of simplicity, we assume that we have enough registers available to allocate vectors A and B at once

6.2.3.2 Scalar Multiplication for Vectors

Now that we have introduced the main elements of deGoal for the building of code generators, we can safely introduce an important feature of our tool: vectorial registers. To do so, we will extend our previous example to scalar multiplication for vectors. Our aim is to compute $[B] = \alpha \times [A]$, where $[A]$ is the input vector, α a scalar known at the time of code generation and $[B]$ is the result vector.

Using standard C, we could write scalar multiplication as in Fig. 6.4. With dynamic code generation, we will specialize the kernel according to the memory location of A, its length, and the value of α. As a consequence, the kernel generated by the compilette will need only one invocation parameter: the address of vector B (assuming that it has the same length than A).

There are several possibilities to implement such a code generator, and we will illustrate two of them here: (1) with the vector support of deGoal instructions (Fig. 6.5) and (2) by mixing `cdg` instructions with plain C to control the code generation and loop over the vector elements (Fig. 6.6). The disassembled binary code that will be produced for these two generated kernel is illustrated in Fig. 6.7a, b,

```
1  void compilette(cdgInsnT* code, int * A_addr, int A_len, int
       alpha) {
2  #[
3    Begin code Prelude B_addr
4    Type  scalar32_t int  32
5    Type  addr_t     uint 32
6    Alloc scalar32_t alpha_r
7    Alloc addr_t     A_addr_r
8    Alloc scalar32_t tmp
9
10   mv alpha_r, #(alpha)
11   mv A_addr_r, #((unsigned int)A_addr)
12 ]#;
13 for (int i=0; i<A_len; i++) {
14   #[
15     lw tmp, @(A_addr_r + #(i))
16     mul tmp, tmp, alpha_r
17     sw @(B_addr + #(i)), tmp
18   ]#;
19 }
20 #[
21   rtn
22   Free tmp, A_addr_r, alpha_r
23   End
24 ]#;
25 }
```

Fig. 6.6 A compilette for scalar multiplication of vectors. The unrolling of the dot product is controlled by C statements

respectively. To use floating-point arithmetic instead of integer, one would simply need to replace `int` by `float` at lines 5 and 6 in Fig. 6.5 and at line 4 in Fig. 6.6, when declaring the types used for scalar arithmetics.

In the compilette illustrated in Fig. 6.5, each of the elements of the vector register `v` will be mapped to a physical register, as long as there are enough registers available on our target processor. In Fig. 6.5, one can see in the generated code that `v` has been mapped on registers `R2` to `R9`. `v` being a vector register of eight elements, the instruction `lw v, tmp` will actually generate eight successive memory loads with a stride of one word from the address contained in the register variable `tmp`, mapped to `R1`. The code generator proceeds similarly for the `mul` and `sw` instructions. The multiplication (`MP`) instruction of the STxP70 processor only works with two register arguments and not with an indirect memory address as an argument. Thus, the instruction `make32 R12, 42` instruction at line 2 of Fig. 6.7a is automatically generated by `mul` to store the contents of variable `A_addr` in the scratch register `R12` and then perform the multiplication operation with register `R12` as an operand.

Figure 6.7a also demonstrates the capability of our instruction scheduler to deal with instruction latency and register dependencies. We will illustrate this point on one example: on the STxP70 processor, the LW instructions have a latency of 3

a

```
1  PUSHRL 0x43F8;;
2  MAKE32 R12, 42; MAKE32 R1, 16176;;
3  LW R2, @(R1+0x0);;
4  LW R3, @(R1+0x4);;
5  LW R4, @(R1+0x8);;
6  LW R5, @(R1+0xC);  MP R2, R2, R12;;
7  LW R6, @(R1+0x10); MP R3, R3, R12;;
8  LW R7, @(R1+0x14); MP R4, R4, R12;;
9  LW R8, @(R1+0x18); MP R5, R5, R12;;
10 LW R9, @(R1+0x1C); MP R6, R6, R12;;
11 SW @(R0+0x0), R2;  MP R7, R7, R12;;
12 SW @(R0+0x4), R3;  MP R8, R8, R12;;
13 SW @(R0+0x8), R4;  MP R9, R9, R12;;
14 SW @(R0+0xC), R5;;
15 SW @(R0+0x10), R6;;
16 SW @(R0+0x14), R7;;
17 SW @(R0+0x18), R8;;
18 SW @(R0+0x1C), R9;;
19 POPRL 0x43F8;;
20 RTS;;
```

b

```
PUSHRL 0x4038;;                          1
MAKE32 R4, 16152; MAKE32 R1, 42;;        2
LW R5, @(R4 + 0x0);;                     3
MP R5, R5, R1;;                          4
SW @(R0 + 0x0), R5;;                     5
LW R5, @(R4 + 0x4);;                     6
MP R5, R5, R1;;                          7
SW @(R0 + 0x4), R5;;                     8
LW R5, @(R4 + 0x8);;                     9
MP R5, R5, R1;;                          10
SW @(R0 + 0x8), R5;;                     11
LW R5, @(R4 + 0xC);;                     12
MP R5, R5, R1;;                          13
SW @(R0 + 0xC), R5;;                     14
LW R5, @(R4 + 0x10);;                    15
MP R5, R5, R1;;                          16
SW @(R0 + 0x10), R5;;                    17
LW R5, @(R4 + 0x14);;                    18
MP R5, R5, R1;;                          19
SW @(R0 + 0x14), R5;;                    20
LW R5, @(R4 + 0x18);;                    21
MP R5, R5, R1;;                          22
SW @(R0 + 0x18), R5;;                    23
LW R5, @(R4 + 0x1C);;                    24
MP R5, R5, R1;;                          25
SW @(R0 + 0x1C), R5;;                    26
POPRL 0x4038;;                           27
RTS;;                                    28
```

Fig. 6.7 Binary code (disassembled) of the kernel generated by the compilettes for scalar multiplication, with alpha = 42. For the sake of simplicity, guard registers are not shown here (**a**) kernel generated from Fig. 6.5 (**b**) kernel generated from Fig. 6.6

cycles. This means that, to avoid cycle stalls, the MP instruction on R2 (line 6) must come three cycles after the instruction `LW R2` (line 3).

The main difference of the C-controlled kernel (Fig. 6.6) with the vectorized kernel (Fig. 6.5) comes from the use of the same register variable `tmp`, mapped on the physical register `R5`. `tmp` successively stores each of the memory loads from vector A (`A_addr`) and is then used to store the result of the multiplication by α (`alpha_r`). Because the same physical register `R5` is used to perform all of the store and multiplication operations for each of the vector elements, our instruction scheduler is not able to bundle the instructions generated in this kernel because of register dependencies. As a consequence, the binary code generated from this kernel (Fig. 6.7b) is far less compact than the code in Fig. 6.7a.

To give an idea of the level of optimization enabled in this example, we compare the execution times of the kernels in Fig. 6.7a, b and of the C version illustrated in Fig. 6.4. The compilation is performed with the `-O3` optimization flag, and the execution times are measured using the simulator of the STxP70 processor in CAS mode, presented later in Sect. 6.3.2.1. The kernels execute, respectively, in 51, 71 and 80 cycles for two vectors containing eight elements. The binary code of the C version counts 18 instruction bundles. This code is even smaller than our kernel in Fig. 6.7a because the C kernel uses the hardware loop instructions of the STxP70. We could help the C compiler with hints about vectorization (e.g., `#pragma unroll`), but unrolling the multiplication at the time of static

compilation is difficult because the vector lengths are not known. On the contrary, at runtime, it becomes easy to perform loop unrolling knowing the lengths of the vector that our kernel will process. For larger loops where unrolling would incur a loss in performance, we could as well use branch instructions and loop structures. This is not shown in this paper for the sake of brevity.

6.3 An Experiment on Matrix Multiplication

6.3.1 Implementation of Matrix Multiplication

This section describes the implementation of a processing kernel for matrix multiplication in order to illustrate the use of deGoal. We describe first a reference implementation, which is statically compiled with the platform's compiler. We then describe two improved implementations using deGoal: the first exploits matrix properties such as matrix size, element size and memory addresses; the second exploits the values of matrix elements.

6.3.1.1 Reference Implementation

Our aim is to perform the standard matrix multiplication as described in equation 6.1, where a, b, and c stand, respectively, for elements of matrices $[A]$, $[B]$, and $[C]$ of sizes $n \times p$, $p \times q$, and $n \times q$:

$$\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, q\}, c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \tag{6.1}$$

The reference implementation of this algorithm is illustrated in Fig. 6.8. We used it as a reference implementation for our experimental measurements.

```
clear(C)
for (y=0; y < n; y++) {
    for (x=0; x < q; x++) {
        for (i=0; i < p; i++) {
            C[x,y] = C[x,y] + A[i,y] * B[x,i]
        }
    }
}
```

Fig. 6.8 Reference implementation of the matrix multiplication (in pseudo C code)

```
/* generation of the kernel's code */
(kernel, v) = compilette(A, B, C)

/* compute matrix multiplication */
clear(C)
kernel();
```

Fig. 6.9 Optimized implementation of the matrix multiplication using deGoal (in pseudocode)

6.3.1.2 First Implementation in a Compilette

A simplified overview of our implementation of the matrix multiplication using deGoal is illustrated in Fig. 6.9. `compilette` is the code generator that produces an optimized kernel function `kernel`, which encompasses the inner-most loop from Fig. 6.8. The code generated for `kernel` depends on the properties of matrices `A`, `B` and `C`: row and column sizes, memory alignment and address of the data in memory. These values are precomputed and propagated into the instructions of `kernel` at code generation time. In consequence, `kernel` does not need invocation parameters.

This implementation of `kernel` is very similar to the reference implementation introduced above, at the exception that all the constants describing matrix properties, which are known at code generation time, have been propagated into the generated code. As we will show in the results section, these improvements alone already contribute to a good performance improvement.

6.3.1.3 Kernel Specialization on Matrix Values

If the matrices to process are sparse or contain remarkable data values, it is possible to further increase performance by specializing the generated code depending on the element *values* of the matrix to process. We illustrate the data-dependent specialization of our processing kernel with a naive algorithm for sparse matrices. Usually, applications that involve the processing of sparse matrices will move to different processing algorithms and to a dedicated representation of data. However, our aim is to illustrate here how, thanks to the use of data-dependent optimizations with runtime code generation, it is possible to drastically improve the performance of our base algorithm.

The code generation is split in two phases (Fig. 6.10): `template_gen` generates the global structure of the processing kernel that is independent of data values in A. At each processing loop, `data_gen` fills the kernel's code upon data values in the row vector to process in A. When there is nothing to execute (e.g., all matrix values in the current row in A are null), `data_gen` returns `NULL` and we immediately move to the next loop step.

This technique involves an extra overhead because the kernel's code is regenerated at each step in the innermost loop. However, as we will show below, this overhead can be compensated very quickly for sparse matrices.

```
clear(C)

/* generate the kernel's structure */
(kernel_templ, v) = template_gen(A, B, C);

/* process matrix multiplication */
for (y=0; y < n; y++){
  for (i=0; i < p; i+=v){
    /* specialize instructions on matrices' data */
    kernel = data_gen(kernel_templ, A, y, i);
    if (NULL != kernel)
      kernel(y, i);
} }
```

Fig. 6.10 Implementation of the matrix multiplication (pseudocode) with code specialization on matrix values

6.3.2 Experimental Results

6.3.2.1 Target Architecture

We target in this work the embedded platform called STHORM (formerly Platform P2012), jointly developed by STMicroelectronics and CEA [2].

The STxP70-4 processor is a 32-bit RISC core. It comes with a variable-length instruction encoding and a dual issue VLIW architecture. Two sets of hardware loop counters are provided to enable loop execution at maximum speed without cycle overheads due to software control. The core processor contains an internal extension for integer multiplication and an optional single-precision floating point extension used in this experiment.

The STHORM SDK is delivered with a full tool chain for compiling, debugging, profiling and simulation in functional and cycle-accurate modes. Our experiments are based on the platform's tool chain and on the ISS simulator of the STxP70 core in CAS (cycle-accurate) mode. In this mode, the simulator models all the latencies that can occur in the processor pipeline: instruction latency, CPU stalls and register dependencies. The latencies of memory accesses are not taken into account by this mode. All our experiments are however using the scratchpad memories (TCDM and TCPM) of the processor, which lowers the effect of this limitation of the simulator in our experiments.

6.3.2.2 Port of deGoal for the STxP70 Processor

deGoal handles by default register allocation and a simple mechanism for instruction scheduling. A simple scheduler allows for the optimization of instruction scheduling with regards to instruction latencies and register dependencies.

However, as compared to standard RISC processors, code generation for the STxP70 processor is a bit more challenging, especially when moving to runtime code generation. Thenceforth, we extended the port of deGoal for this architecture with VLIW support: optimizing the dual issue and the construction of instruction bundles. Also the floating-point support comes as an extension and uses a separate register file of 16 32-bit registers. Our port of deGoal supports all of these features of the STxP70.

6.3.2.3 Experimental Setup

We have evaluated our optimized version of the matrix multiplication against the reference implementation described in Sect. 6.3.1.1.

The reference implementation is compiled in `-O3`. Loop unrolling and support of hardware loop counters and of the floating-point extension are also enabled. The best performance for the reference implementation was eventually obtained with an implementation close to the pseudocode described in Fig. 6.8, with the addition of `#pragma unroll 8` on top of the innermost loop.

The code generated by deGoal's compilette does not depend on compiler optimizations, because it is generated at runtime by the compilette. Hence whatever the compiler optimizations selected, the execution time of the generated kernel remains constant. Compiler optimizations have however an effect on the performance of the compilette, because it is statically compiled as a standard application component. In our performance measurements, we have used the same compiler options to compare the reference implementation and our implementation using deGoal.

We have also exploited the VLIW extension of the STxP70-v4 core, using the appropriate compilation flags. On the compilette's side, VLIW support is integrated in the `cdg` pseudo-ASM language of deGoal. As a consequence, it is not exposed to the developer and the compilette is tailored to automatically exploit this feature as soon as the processor supports it.

6.3.2.4 Measure of the Code Generation Time

We have instrumented the compilette to measure the time spent in code generation at runtime: code generation takes from 150 to 300 cycles per instruction bundle generated. The variation of the average speed of code generation per instruction bundles is due to the computation of instruction bundling, the scheduling of instructions according to register dependencies and the extra computations done at the end of code generation, for example, computing the jump addresses. The best code generation speed is achieved for unrolled code without instruction bundling.

The code generation time is not taken into account in the speedup results presented below, because it is not necessary to regenerate the code for each matrix multiplication. As an indicator, code generation represents 100% of the execution time for a multiplication of 16×16 matrices and less than 0.1% for 256×256 matrices.

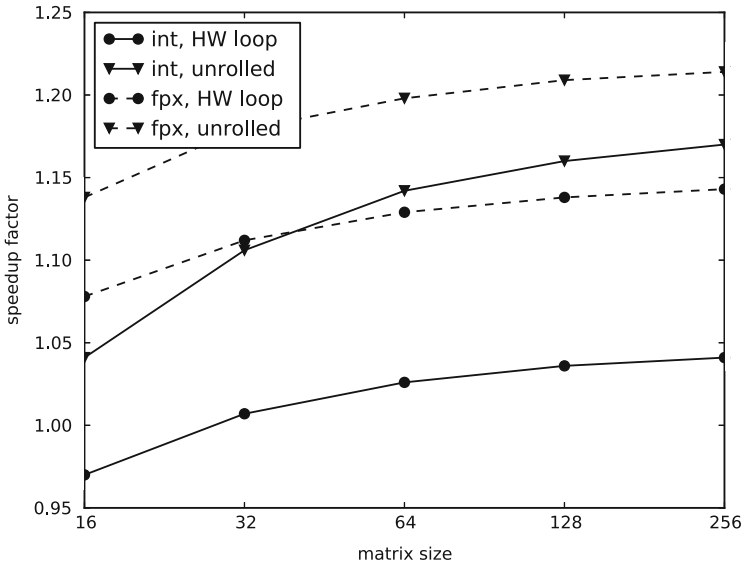

Fig. 6.11 Speedup factor measured, for integer multiplication (*plain line*) and floating-point multiplication (*dashed line*), according to the implementation described in Sect. 6.3.1.2

6.3.2.5 Performance of the Processing Kernels

Figure 6.11 illustrates the performance improvements achieved using deGoal as compared to the reference implementation compiled with full optimization, for two cases of code generation: using the hardware loop counters provided by the STxP70 core (`HW loop`) and fully unrolling the kernel's code (`unrolled`). The speedup factor s represents the reduction factor of the execution duration of our implementation as compared to the reference implementation. We calculate it as follows: $s = \frac{t(\text{ref})}{t(\text{degoal})}$, where $t(\text{ref})$ measures the time execution of the reference implementation and $t(\text{degoal})$ the time execution of the generated kernel. Our implementation using compilette brings a good overall performance improvement: when the matrix size is 256×256 elements, we achieve a reduction of the execution time of 21% for integer multiplication and of 17% for floating-point multiplication.

Figure 6.12 illustrates the speedup factor measured when using code specialization on the data of matrix A, as presented in Sect. 6.3.1.3. We illustrate here the most favorable case where matrix A is the identity matrix. In this case, the looped implementation shows a huge speedup because of the instructions removed from the kernel when null values are met in matrix A. The unrolled version is not efficient, considering the favorable experimental conditions, because a part of the code generation is performed *during* kernel's execution, and code unrolling requires a lot more instructions to be generated.

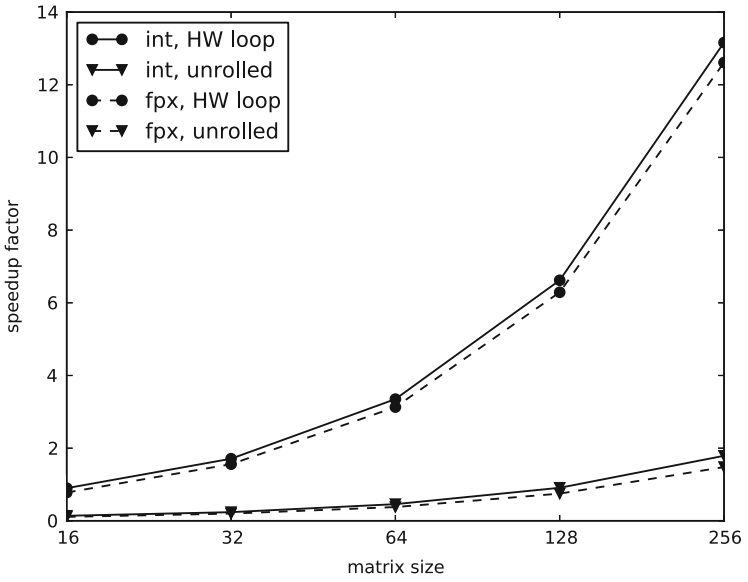

Fig. 6.12 Speedup factor measured, for integer multiplication (*plain line*) and floating-point multiplication (*dashed line*), according to the implementation described in Sect. 6.3.1.3

6.4 Related Work

There is an extensive amount of literature about approaches related to our work with deGoal.

Dynamic compilation and interpretation are most of the time used together in just-in-time compilers (JITs) [1]. JITs use interpretation for the parts of the program that are run seldom, and dynamic compilation is reserved for hotspots, which are identified by tracing the application activity at runtime. Such techniques usually require embedding a large amount of intelligence in the JIT framework, which means a large footprint and a significant performance overhead. In order to target embedded systems, some research works have tried to tackle these limitations: memory footprint can be reduced to a few hundreds of KB [5], but the binary code produced often presents a lower performance because of the smaller amount of optimizing intelligence embedded in the JIT compiler [6].

In deGoal, the objective is to reduce the cost incurred by runtime code generation. Our approach allows generating code at least ten times faster than traditional JITs: JITs hardly go below 1,000 cycles per instruction generated while we obtain 25 to 80 cycles per instruction generated on the STxP70 processor. Our approach is similar to partial evaluation techniques [7,8], which consists in precomputing during the static compilation passes the maximum of the generated code to reduce the runtime overhead. In partial evaluation, the machine code is finalized at runtime by selecting code templates and filling pre-compiled binary code with data values

and jump addresses. Using deGoal we compile statically an *ad hoc* code generator (the compilette) for each kernel to specialize. An originality of our approach relies in the possibility to perform runtime instruction selection depending on the *data* to process [3].

Code specialization is a technique similar to partial evaluation. Specialization can be done statically, at compilation time, or dynamically. C++ templates might be the most well-known static specializer. The developer writes a function that is parametrized by a list of types or integer-constant parameters. When the template is used, the user indicates missing parameters and the compiler automatically generates the new function according to those parameters. This brings more flexibility during the development process at the expense of a fast growing binary size, because for each set of parameters, a new function is generated. However, the template parameter values have to be known at compile time, which strongly limits the number of code optimizations applicable.

Equivalent systems that operate at runtime are less used and include a larger diversity of approaches, which can be regrouped into different categories. Fully manual approaches rely on the user to describe what should be generated at runtime [9, 10]. With this approach, the user has a fine control over the generated code. Semi-manual approaches rely on the user to annotate parameters that should be specialized [11]. Fully automatic approaches try to detect, at compile time, the code parts that could benefit from runtime code generation [12–14]. Each approach avoids an explosion in code size while maintaining a larger spectrum in which it can be used. A major advantage is the capability to cover the whole function application domain without having to speculate on parameter values. The major drawback is, of course, that a part of the compilation cost has to be paid at runtime and has to be amortized in one way or another.

Various optimizations can be performed in the various times of application lifetime, for instance, during static compilation (where most optimizations are usually performed), during link time [15, 16] or during installation time like ATLAS [17]. Some tools use complex schemes like FFTW [18], where multiple code variants are generated, compiled and then evaluated at installation time. Then, during program initialization, the codelets are selected by a planner that is parametrized by the size of the DFT to be computed. Another example that use a similar approach is ATLAS [17]. Some other optimizations are staged across several times. For instance, iterative compilation accumulates information through different times. With enough information, it rolls back to an earlier time to perform new optimizations. Profile-guided compilation (PGC) uses execution traces obtained by running the application with a learning data set to perform new optimizations. But few tools are able to perform optimization based on the runtime environment.

Late code specialization is very close to our approach. Generally speaking, these approaches pre-compile statically a template version of the application code, which is completed at runtime by a code specializer. `C [10] extends the C syntax by adding syntactic elements like ` or @ to describe parts of code that will be generated at runtime. The compilation phase transforms `C expressions into an intermediate representation (IR). At runtime the IR is assembled and compiled via simplified

compiler back-end. DyC [11] is a tool that creates code generators from an annotated C code. Like `C, it adds some tokens such as @ to evaluate C expressions and inject the results as an immediate value into the machine code. Calpa [14] uses profile-guided compilation to detect functions that could benefit from runtime code specialization and generate the code generator using DyC. Tempo [13] works on an unannotated subset of C. It analyzes the source code to detect parts of the code that could benefit from constant propagation and creates a binary template from it. At runtime, the template is filled with missing values and executed. Fabius [12, 19] uses a similar approach to Tempo, applied to ML. Our approach differs from those tools targeting late code specialization by using:

- A low-level code representation with vector description
- No manipulation of bytecode at runtime
- The capability to control the code generation
- The capability to perform cross-architecture code generation.

Approaches for multicore architectures mostly use a classical JIT information. LLVM [20] (low level virtual machine) is a compilation framework that can target many architectures, including x86, ARM or PTX. One of its advantages is the unified internal representation (LLVM IR) that encodes a virtual low-level instruction with some high-level information embedded on it. Various tools were built on top of it, starting with clang, a C/C++/Objective-C compiler. With the release of the CUDA toolkit 4.1, the Nvidia compiler is based on LLVM. The driver loads a textual representation of the assembly language targeting the GPU and then dynamically compiles it to a binary representation. This technique here is mainly used to hide the implementation details of Nvidia GPUs and not in the purpose of runtime optimizations. Our approach avoids the use of bytecode manipulation to focus on specialization using runtime information. In multicore architectures, the lightness of our approach enables self-generation capabilities at the level of processing elements, and compilettes can perform cross-code generation in heterogeneous architectures.

6.5 Conclusion

In this paper, we have introduced deGoal as a tool for runtime code generation, thanks to the integration of compilettes in a binary application, and we have illustrated the benefits of using deGoal to optimize processing kernels.

We have shown that deGoal can easily compete with a highly optimized code produced by a static compiler with little effort: the code produced has better performance than a code statically compiled with full optimization, and furthermore the quality of the code produced with deGoal is consistent and does not depend on compiler's options. deGoal also allows to specialize the code of a processing kernel for a particular set of runtime data, which is not possible using a static compiler.

We have shown that for processing kernels with a high dependency on the data to process the performance increase can be huge.

Because deGoal is related to the generation of machine binary instructions, its scope of application is actually restricted to the processor. In order to use these optimization techniques in large-scale platforms, e.g., MPSoCs or HPC clusters, one must rely on tools of higher level for the parallelization of an application on multiple processing elements. Future work will present how it is possible to integrate kernels optimized with deGoal compilettes in large-scale applications.

deGoal is currently under active development. It is able to produce code for multiple platforms: Nvidia GPUs, ARM processors (Jazelle, SIMD, Thumb, NEON), the STxP70, and other RISC processors under NDA.

References

1. J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, pp. 97–113, June 2003.
2. P2012DAC12 T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the java hotspot client compiler for java 6," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 1, pp. 7:1–7:32, May 2008.
3. H.-P. Charles, "Basic infrastructure for dynamic code generation," in *workshop "Dynamic Compilation Everywhere", in conjunction with the 7th HiPEAC conference*, H.-P. Charles, P. Clauss, and F. Pétrot, Eds., Paris, France, january 2012.
4. D. Couroussé and H.-P. Charles, "Dynamic code generation: An experiment on matrix multiplication," in *Proceedings of the Work-in-Progress Session, LCTES 2012*, June 2012.
5. A. Gal, C. W. Probst, and M. Franz, "HotpathVM: an effective JIT compiler for resource-constrained devices," in *VEE '06*. New York, NY, USA: ACM, 2006, pp. 144–153.
6. N. Shaylor, "A just-in-time compiler for memory-constrained low-power devices," in *Java VM'02*. Berkeley, CA, USA: USENIX Association, 2002, pp. 119–126.
7. C. Consel and F. Noël, "A general approach for run-time specialization and its application to C," in *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, 1996, pp. 145–156.
8. N. D. Jones, "An introduction to partial evaluation," *ACM Comput. Surv.*, vol. 28, pp. 480–503, September 1996.
9. K. Brifault and H.-P. Charles, "Efficient data driven run-time code generation," in *Proc. of Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Houston, Texas, USA, Oct. 2004.
10. D. R. Engler, W. C. Hsieh, and M. F. Kaashoek, "'C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation," in *In Symposium on Principles of Programming Languages*, 1996, pp. 131–144.
11. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, "DyC: an expressive annotation-directed dynamic compiler for C," *Theor. Comput. Sci.*, vol. 248, no. 1–2, pp. 147–199, Oct. 2000. [Online]. Available: http://dx.doi.org/10.1016/S0304-3975(00)00051-7
12. M. Leone and P. Lee, "Lightweight Run-Time Code Generation," Department of Computer Science, University of Melbourne, Tech. Rep., 1994.
13. C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé, "Tempo: specializing systems applications and beyond," *ACM Comput. Surv.*, vol. 30, no. 3es, Sep. 1998. [Online]. Available: http://doi.acm.org/10.1145/289121.289140

14. M. Mock, C. Chambers, and S. J. Eggers, "Calpa: a tool for automating selective dynamic compilation," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: ACM, 2000, pp. 291–302. [Online]. Available: http://doi.acm.org/10.1145/360128.360158
15. B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "Plto: A link-time optimizer for the intel ia-32 architecture," in *In Proc. 2001 Workshop on Binary Translation (WBT-2001*, 2001.
16. B. De Sutter, B. De Bus, and K. De Bosschere, "Sifting out the mud: low level c++ code reuse," *SIGPLAN Not.*, vol. 37, no. 11, pp. 275–291, Nov. 2002. [Online]. Available: http://doi.acm.org/10.1145/583854.582445
17. R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *SuperComputing 1998: High Performance Networking and Computing*, 1998. [Online]. Available: http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps
18. M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the FFT," in *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, vol. 3, Seattle, WA, May 1998, pp. 1381–1384. [Online]. Available: citeseer.ist.psu.edu/frigo98fftw.html
19. M. Leone and P. Lee, "A Declarative Approach to Run-Time Code Generation," in *In Workshop on Compiler Support for System Software (WCSSS*, 1996, pp. 8–17.
20. C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 2002.

Part IV
Case Studies

Chapter 7
Signal Processing: Radar

Michel Barreteau and Claudia Cantini

7.1 Brief Description of the RT-STAP Algorithm

Space-time adaptive processing (STAP) is a processing technique operating in the space-time domain that allows the simultaneous cancellation of clutter and jamming via the computation of a 2D cancellation filter.

Essentially, the radar is required to have an array (for instance, a linear array along the aircraft axis) of L antennas each receiving K echoes from a transmitted train of K coherent pulses PRT (pulse repetition time) seconds far apart. The STAP filter operates the simultaneous processing of the spatial samples, i.e., the different channels from different antenna elements, and the temporal samples collected from multiple (consecutive) pulses of the transmitted train. Processing data from multiple channels enables the control of the directional response of the system, while processing data from multiple pulses enables the separation of signals based upon their Doppler frequencies. Unwanted received signals, such as ground clutter and jamming signals, can thereby be efficiently suppressed. The STAP application comprises calculations of adaptive weights (w) and application of these weights on the input data (vector x) (Fig. 7.1). Under the hypothesis of disturbance having a Gaussian probability density function and a target with a certain Doppler frequency and direction of arrival, the output signal of the optimum processor is provided by the linear combination of the LK echoes x (vector representing the physical data cube at a certain range cell under test, CUT) with weights $w = M^{-1}s^*$. M is the noise covariance matrix estimation, i.e., $M = E\{x^*x^T\}$ where x

M. Barreteau
Thales Research & Technology, Campus Polytechnique - 1 avenue Augustin Fresnel, 91767, Palaiseau Cedex, France
e-mail: michel.barreteau@thalesgroup.com

C. Cantini (✉)
Selex ES, Via Tiburtina Km. 12,400, 00131, Roma RM, Italy
e-mail: claudia.cantini@selex-es.com

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_7,
© Springer Science+Business Media New York 2014

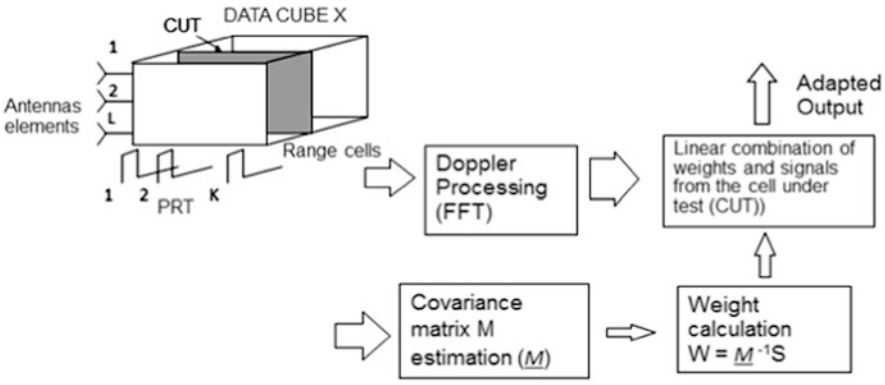

Fig. 7.1 Simplified schema of the real-time STAP radar application

(dimension $LK \times 1$) is the collection of the LK disturbance echoes in a range cell, and s—the space-time steering vector—is the collection of the LK samples expected by the target. The superscripts * and ^{T} stand, respectively, for complex conjugate and transpose. Such a processing technique is highly demanding in terms of computational power requesting the covariance matrix inversion for the weight calculation [1].

From the computational viewpoint, the input data structure is a data cube, i.e., a stream, of thousands of vectors of typically 512 or 1,024 complex single-precision numbers that are used for calculating the covariance matrix estimation. The most critical phase (bottleneck) in the application workflow is the weight calculation through covariance matrix inversion, i.e., the resolution of a very large system of N linear equations on complex numbers, of order $O(N^3)$. This computation is applied to each covariance matrix calculated from each vector belonging to the data cube, according to a streamlike behavior. As performance measures, we are interested mainly in the throughput parameter, i.e., the average number of computed cubes per second. Due to the computational characteristics of this application, this throughput measure is obtained by the evaluation of the service time per matrix. On a current machine, the sequential execution of a system of linear equations on complex numbers, of the sizes indicated above, has a service time per matrix equal to about 10^2–10^4 ms for single-precision numbers. For a true real-time exploitation of STAP, our target service time per matrix is required to be of the order of 10^0–10^1 ms; thus our requirement is a performance improvement of two orders of magnitude to be achieved through parallelization strategies. This goal can be met only if we are able to find almost linear scalable solutions for the parallelization (not a simple task for our target problem due to the large problem size and to the heavy data dependencies in a nested loop computation). With lower priority, we are also interested in the latency per matrix. In some applications we could also accept a latency of the same order of magnitude of the sequential version, while in other cases a sensible latency decrease could be required too. All the other phases of the

applications, though potential and interesting candidates for parallelization (e.g., Doppler processing), have much less stringent performance requirements (they are at least one order of magnitude faster than the weight calculation) and, nowadays, they can be implemented according to very efficient sequential algorithms and libraries (e.g., FFT). However, in massively parallel implementation of the whole application, also these phases could become candidates for parallelization [1].

7.1.1 Detailed Description of the Computational Phases

In the following, for the reasons discussed above, we will consider the weight calculation phase only. For the resolution of the linear system of equations, we use the Cholesky factorization direct method, which applies correctly to all the occurrences of this problem in STAP applications and is characterized by lower complexity compared to other direct methods (QR versions). For our purposes, the Cholesky factorization is considered the application bottleneck. As discussed above, the Cholesky factorization operates on matrices of 512×512 or $1,024 \times 1,024$ single-precision complex numbers organized in streams, where the generic stream element is a vector of 512 or 1,024 single-precision complex numbers.

The classical Cholesky factorization transforms a hermitian positively defined matrix A into the product of a lower triangular matrix L and of its conjugate transpose upper triangular matrix L^T:

$$A = LL^T$$

The basic algorithm applies the method definition directly. It is described by the following algorithmic pseudocode to generate matrix L from matrix A:

```
for (j = 0; j < n; j++) {
    sum = 0;
    for (k = 0; k < j; k++) {
        sum += L_jk^2;
    }
    L_jj = sqrt(A_jj − sum);
    for (i = j + 1; i < n; i++) {
        sum = 0;
        for (k = 0; k < j; k++) {
            sum += L_ik * L_jk;
        }
        L_ij = (A_ij − sum)/L_jj;
    }
}
```

In this nested control structure the size of data structures (notably, parts of columns) varies at every computation step, though according to a statically recognizable and predictable pattern. The literature contains several alternative versions of the basic sequential algorithm for Cholesky factorization. Some block-based versions have been studied in order to optimize the locality and reuse properties of matrix parts accessed during the various computation steps. A is represented as composed of smaller square blocks. This can improve the locality and reuse exploitation: though paid in terms of a larger number of elementary operations on matrix blocks, these properties are the key for potential optimizations of memory hierarchy structures, especially in architectures where caching is not primitive. The matrix representation can be the following:

$$\begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} * \begin{pmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{pmatrix}$$

where

$$A_{11} = L_{11} * L_{11}^T$$

$$A_{21} = L_{21} * L_{11}^T$$

$$A_{21}^T = L_{11} * L_{21}^T$$

$$A_{11} = L_{21} * L_{21}^T + L_{22} * L_{22}^T$$

Denoting $chol$ the application of the Cholesky basic algorithm:

$$L_{11} = chol(A_{11})$$

and iteratively

$$L_{21} = A_{11}/L_{11}^T$$

$$L_{22} * L_{22}^T = A_{22} - L_{21} * L_{21}^T$$

$$A_{22} = L_{22} * L_{22}^T$$

$$L_{22} = chol(A_{22})$$

An algorithmic pseudocode for the block method applied to the Cholesky factorization is (B denotes the number of blocks composing the original matrix):

```
for(k = 0 to B) do {
    L_kk = chol(A_kk);
```

```
    L_kk^T = transpose(L_kk);
    L_kk^-T = invert(L_kk^T);
    for (i = k + 1; i < B; i++) {
        L_ik = A_ik * L_kk^-T
    }
    for (j = k+1; j < B; j++) {
        L_jk^T = transpose(L_jk);
        for (i = j; j < B; i++) {
            A_ij = A_ij - L_ik * L_jk^T
        }
    }
}
```

Here we can statically recognize different patterns for data dependencies with respect to other versions not operating on blocks.
According to the application environment, it is possible that the application operates on cubes that are produced in storage subsystems accessible through I/O and/or files. Owing to the order of magnitudes of the calculation times, the I/O latency for a cube transfer can be overlapped with the internal calculation of previous cubes [2, 3].

7.1.2 Data-Parallel Cholesky Factorization

The block Cholesky algorithm can be expressed using several methods, but two of them are the most used: the left-looking method and the right-looking method. Both methods use the same kernel subroutine to do the numerical work. The differences are mainly in the memory access pattern and in cache data locality exploitation. From the parallelization point of view, the right-looking version expresses more parallelism as the algorithm explores the data dependency graph breadth-first, whereas the left-looking version is less parallel but more cache-oblivious.

In the following we outline in detail the communication pattern of the block-based right-looking version for the Cholesky factorization.

Suppose an $N \times N$ input matrix A. By choosing a block size of m, the input matrix can be split in a set of $\frac{N}{m} \times \frac{N}{m}$ blocks. Exploiting maximum parallelism in the computation of the Cholesky algorithm, each block can be computed by a distinct virtual processor, i.e., a concurrent entity which can be executed on an abstract machine. This results in a set of virtual processors (VPs), which cooperate in getting the factorization done by using explicit messages to resolve data dependencies (for the sake of simplicity we assume a message-passing abstract machine). Without any loss of generality, we also assume that the input matrix is distributed onto the set of VPs row-wise, i.e., the VP_{ij} owns the matrix block data L_{ij} in his local virtual memory.

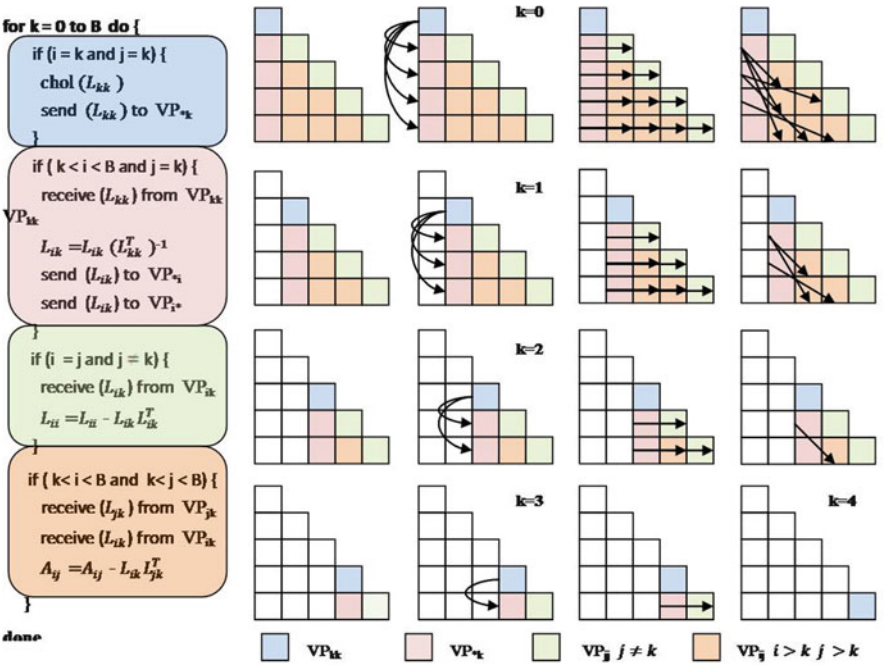

Fig. 7.2 The pseudo code of the generic VP ij (*left*)

The pseudocode of the generic abstract executor VP_{ij}, and the communications among VPs at each steps for the case $N = 5$, is sketched in the following figure ($B = \frac{N}{m}$).

The abbreviated notation VP_{*k} informally stands for "all valid VPs in the column k"; in the same way VP_{k*} stands for "all valid VPs in the row k".

At each step of the main **for** loop, it is possible to identify four sets of distinct VPs, each one with different data dependencies (i.e., activation condition). In Fig. 7.2 we represent different sets with different colors.

At the k-th step, first the VP_{kk} is executed by using only the local block data values, and then the resulting block L_{kk} is sent to all VPs in the k-th column; thus, upon receiving the block, each VP_{*k} can be executed in parallel. The resulting L_{ik} block produced by the VP_{ik} is then sent in parallel to all VPs in the sets VP_{*i} and VP_{i*}, i.e., to all VPs whose row index and column index is equal to the current VP row index i. The VPs in the right-side submatrix of size $[N - (k + 1)]$, upon receiving the blocks, update their local value A_{ij}. It is worth noting that no explicit barrier is needed to synchronize different sets of VPs during external loop iterations.

As the k index approaches B, the number of computing VPs decreases and the communications stencil changes his shape (thus changing the number and the size of the communications). A critical aspect to take into account is the mapping of the

VPs onto the physical processors (generally a subset of the number of VPs) or, in other words, the data distribution of the input matrix [4, 5].

7.2 Related Tool-Chain

The demonstrator leverages a tool-chain which includes:

- A front-end tool: SpearDE by Thales Research & Technology, which offers a graphical interface for describing computation kernels interacting in a data-flow model. It also allows to graphically model the target platform and the mapping of the computation kernels onto this platform.
- A back-end tool: Par4All by SILKAN, which is a source-to-source parallelizing compiler able to provide automatic parallelization of loops.
- A MCAPI layer by CEA which provides a number of low-level tools and runtimes for thread management, memory allocation, and communication.

The tool-chain used for the demonstrator is sketched in Fig. 7.3. It consists in coupling SpearDE [6] (front-end tool), Par4All [7] (back-end tool) and a MCAPI runtime. SpearDE is a graphical model-based design environment which provides the user with both domain-specific application interfaces and heterogeneous execution platform description interface in order to help the implementation of data-streaming applications on parallel machines. Par4All is a source-to-source parallelizing tool generating tasks or parallel programs for various targets from sequential code (C, Fortran, Scilab, Matlab). The MCAPI runtime is built on top of STHORM.

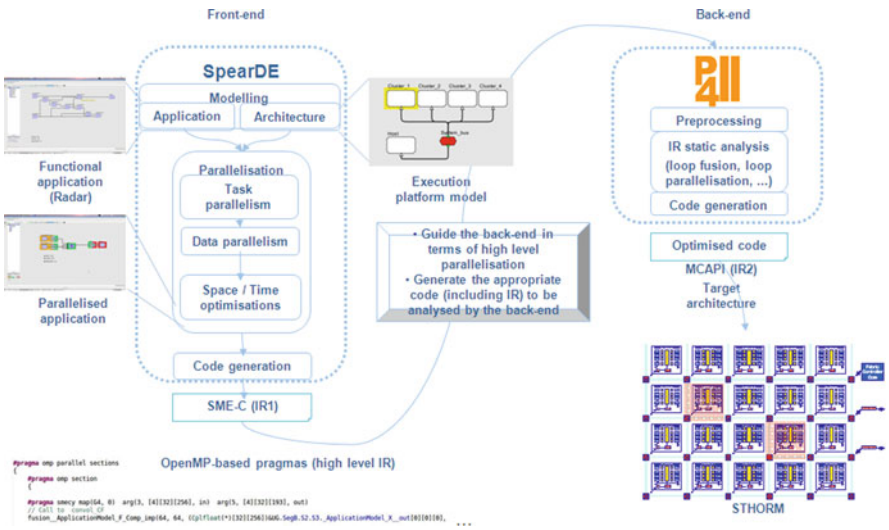

Fig. 7.3 Tool-chain used for the demonstrator

SpearDE allows rapid prototyping on the STHORM platform having the possibility to generate the SMECY IR1 (called SME-C). SpearDE is suited for regular data-streaming applications. It easily handles multidimensional arrays to partition and distribute them onto the STHORM memories. Starting from IR1, the Par4All tool is used to find parallelism in the computational kernels by discovering parallel loop nests through data dependency analysis. This is done by relying on the PIPS open source project [8] which is able to analyze the effects of the program operations by using an abstract interpretation. Parallel loop nests can be replaced by a corresponding kernel call on the low-level platform. The MCAPI layer provides thread management, memory allocation and communication primitives.

The main design steps in this tool-chain are the following ones:

Modeling Both the application (RT-STAP) and the execution platform (STHORM) have to be graphically modeled. The real-time STAP application will be built from scratch here but SpearDE also accepts C99-based code with a set of coding rules.

Parallelization First it consists in allocating computing tasks to hardware resources. Then data parallelism is pointed out. Finally communication tasks will be inserted where needed and scheduling may be refined.

Code generation All the previous parallelization steps are exploited to generate the appropriate IR1 code. This guides Par4All to generate an efficient IR2 code through several passes.

Execution The IR2 primitives rely on the MCAPI layer to ensure an efficient execution on STHORM.

7.2.1 Application and Execution Platform Modeling

The application model of the demonstrator is shown in Fig. 7.4. The block Cholesky algorithm (green dashed lines around the *Chol* module) has been decomposed into several nodes. Each node in the graph matches a computing task (one step of the Cholesky algorithm) that includes a (parallel) loop nest; it executes a so-called elementary task that is usually iterated by several static affine nested loops. The elementary task represents a basic operation (e.g., matrix multiply or inversion, convolution) or a function already optimized or inherently sequential. The *Cov* module (in blue) and the *Wei* module (in orange) were not decomposed in multiple elementary nodes. The first and last tasks are artificial tasks that, respectively, generate inputs and test results.

The SpearDE model of the STHORM platform is shown in Fig. 7.5. The platform model gives a hierarchical representation of the target system which is not the exact representation of the physical platform (the 16 cores of each cluster are not detailed because the generated code will be executed at cluster level but the

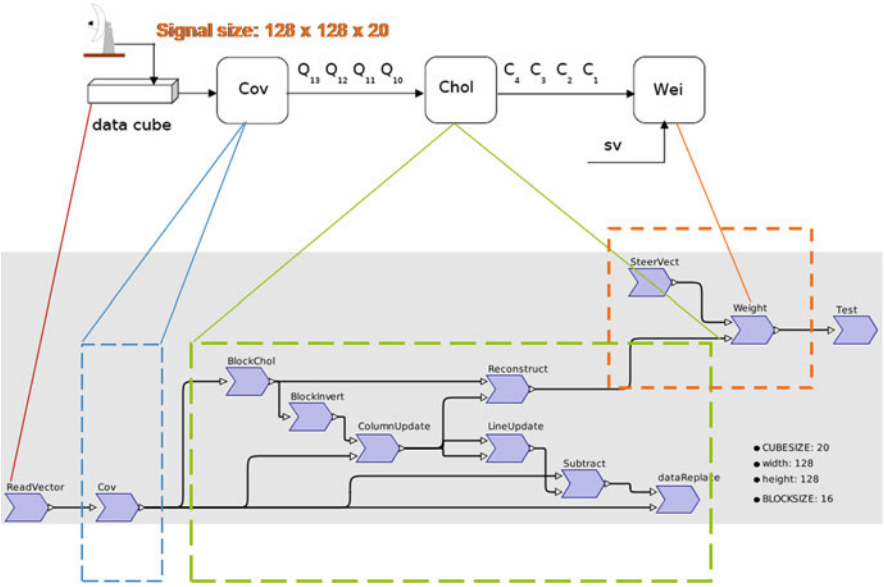

Fig. 7.4 SpearDE application model

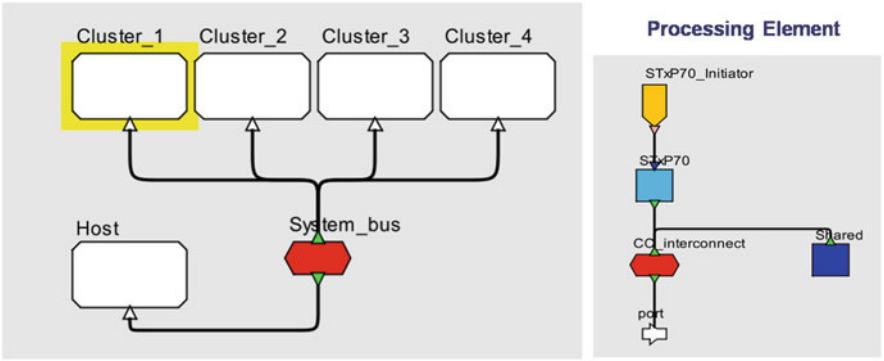

Fig. 7.5 SpearDE model of the STHORM platform

number of cores is known under the form of an attribute) but is detailed enough to allow automatic generation of communication nodes in case of distributed memory data accesses or when data reorganization is needed between two existing nodes. SpearDE relies on the described topology to compute communications between the different memories.

Due to STHORM cluster memory constraints and considering the proposed parallelization in SpearDE, the maximum input size for the demonstrator input data cube is $128 \times 128 \times 20$ complex float numbers (total size of 2.5 MB). The block size

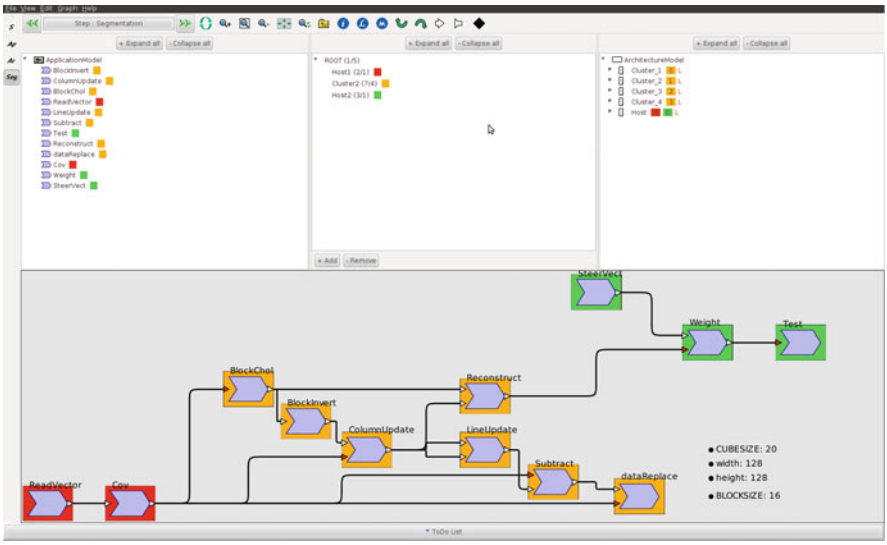

Fig. 7.6 Mapping of different application modules in SpearDE

is set to 16×16 elements in order to have a good balance between parallelism and computation granularity.

7.2.2 Parallelisation on the STHORM Platform

Considering the STHORM platform which includes four clusters, each one with 16 processing elements and 256 KB of local shared memory, the following mapping of modules has been applied:

- The *Cov* module is mapped on the host processor (the red coloured modules at the bottom of Fig. 7.6).
- All modules composing the block Cholesky factorization (*Chol*) are replicated on each cluster of the STHORM platform in order to be able to compute four matrices at a time (all orange-colored modules).
- The *Wei* module is mapped on the host processor working in pipeline with the 4 *Chol* modules (the green-colored modules).

This allocation is quickly done thanks to a graphical interface (through drag and drops). At the top of Fig. 7.6, computing tasks (flattened view) are listed on the left-hand side and hardware resources on the right-hand side. An association (depicted in the center by a color) between a subset of computing tasks and a part of available computing resources means that these tasks will be executed by the selected hardware resources (same color).

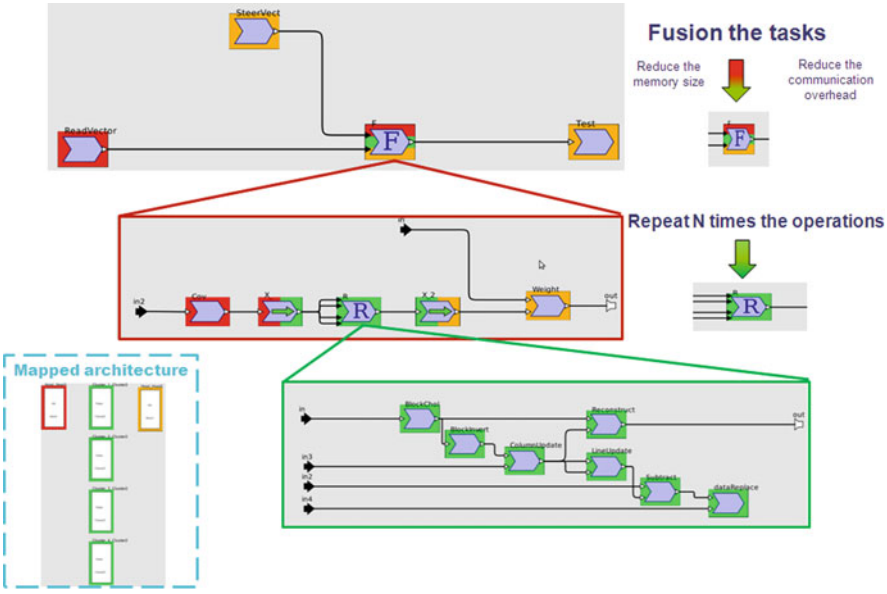

Fig. 7.7 Hierarchical view of the parallelized application under SpearDE

Once this task parallelism decided, the user has to repeat the *Chol* tasks 16 times to cover the whole matrix (through a push-button mechanism again). The next step consists in operating a task fusion around these tasks (including the communication tasks) to implement a round-robin distribution. This task fusion optimizes the memory occupancy (depending on the life duration of arrays) that fits the STHORM memory sizes. Communications between the Host and the STHORM are automatically inserted by pressing a button. This is done in order to send input data to the accelerator and to receive results back from STHORM.

The resulting parallelized application is shown in Fig. 7.7.

7.2.3 IR Code Generation

7.2.3.1 SpearDE/IR1 and Par4All/IR2

SpearDE translates the results of these parallelization steps into an IR1 code. The related Fig. 7.8 shows:

- Parallel loops (`#pragma omp parallel`) with other OpenMP directives (e.g., `schedule(static,1)` means 1 thread per processor with a static scheduling)
- Some SMECY-specific mapping directives (`#pragma smecy map`)

- The round-robin distribution (`num_threads(4) schedule(static,1)`)
- Some communication primitives (`#pragma smecy communication`) that make this IR1 executable.

All these informations enable to guide Par4All in producing an efficient IR2 code. As Par4All is a source-to-source compiler, it refines this SME-C code through several passes: it uses SMECY-specific macros (see below `SMECY_*`)

Listing 7.1 SMECY-specific macros generated by Par4All

```
void smecy_func_fusion_F_F2_fft_CF_6() {
 SMECY_set(fusion_F_F2_fft_CF,6,STHORM,1,0);
 SMECY_send_arg_vector(fusion_F_F2_fft_CF,1,Cplfloat,((Cplfloat (*)[32UL])(UG -> SegClusters.S2.S3.
 F_X_4_out[0]
 + 0)),193 * 32,STHORM,1,0);
 SMECY_prepare_get_arg_vector(fusion_F_F2_fft_CF,2,Cplfloat,((Cplfloat (*)[32UL])(UG ->
        SegClusters.S2.S3.F_Filt_Dop_out[0] + 0)),193 * 32,STHORM,1,0);
 SMECY_launch(fusion_F_F2_fft_CF,2,STHORM,1,0);
 SMECY_get_arg_vector(fusion_F_F2_fft_CF,2,Cplfloat,((Cplfloat (*)[32UL])(UG -> SegClusters.S2.S3.
  F_Filt_Dop_out[0] + 0)),193 * 32,STHORM,1,0);
 SMECY_cleanup_send_arg_vector(fusion_F_F2_fft_CF,1,Cplfloat,((Cplfloat (*)[32UL])(UG -> SegClusters.S2.S3.
  F_X_4_out[0] + 0)),193 * 32,STHORM,1,0);
 SMECY_accelerator_end(fusion_F_F2_fft_CF,6,STHORM,1,0);
 // Call to fusion_F_F2_fft_CF }
```

that are translated into SMECY MCAPI primitives (see `SMECY_MCAPI_*`).

```
// Call to Vect2Cov
#pragma smecy map(Host)  arg(3, [128][400], in)  arg(5, [21][128][128], out)
Vect2Cov(128, 400, (Cplfloat(*)[400])&UG->SegHost.S1.ReadVector_out[0][0], 21, (Cplfloat(*)[128][128])&UG->SegHost.S1.Vect2Cov_out[0][0][0], 200, 10);

int i_F_0;
#pragma omp parallel for private (i_F_0,idxTime) num_threads(4) schedule(static,1)
for (i_F_0 = 0; i_F_0 < 21; i_F_0++) {

    threadprivate_i_F_0 = i_F_0;
    #pragma omp critical
    {
    // Call to BlockCov
    #pragma smecy map(Host)  arg(4, [128][128], in)  arg(5, [16][16][8][8], out)
    BlockCov(128, 8, 16, (Cplfloat(*)[128])&UG->SegHost.S1.Vect2Cov_out[0 + i_F_0*1][0][0], (Cplfloat(*)[16][8][8])&UG->SegHost.S1.S2.F_BlockCov_out[0][0][0][0]);

    // Communication from x86.External_DDR to Cluster.Shared
    #pragma smecy communication src(External_DDR,Host) dst(Shared,STHORM,(threadprivate_i_F_0)%4,0)
    memcpy(&UG->SegClusters[(threadprivate_i_F_0)%4].S3.F_X_2_out[0][0][0][0], &UG->SegHost.S1.S2.F_BlockCov_out[0][0][0][0], 16*16*8*8*sizeof(Cplfloat));

    }
    int i_F_R_0;
    idxTime = 0;
    for (i_F_R_0 = 0; i_F_R_0 < 16; i_F_R_0++) {

        // Call to ComputeCholesky
        #pragma smecy map(STHORM,(threadprivate_i_F_0)%4,0)  arg(2, [8][8], in)  arg(3, [8][8], out)
        ComputeCholesky(8, (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.F_X_2_out[0][0][0][0], (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockChol_out[0][0]);

        // Call to Invert
        #pragma smecy map(STHORM,(threadprivate_i_F_0)%4,0)  arg(2, [8][8], in)  arg(3, [8][8], out)
        Invert(8, (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockChol_out[0][0], (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockInvert_out[0][0]);

        // Call to fusion_F_R_F_ColumnUpdate
        fusion_F_R_F_ColumnUpdate(8, (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockInvert_out[0][0], (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4].S3.F_X_2_

        // Call to fusion_F_R_F3_LineUpdate
        fusion_F_R_F3_LineUpdate(8, (Cplfloat(*)[8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_ColumnUpdate_out[0][0][0], (Cplfloat(*)[8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F

        // Call to Reconstruct2
        #pragma smecy map(STHORM,(threadprivate_i_F_0)%4,0)  arg(3, [8][8], in)  arg(4, [16][8][8], in)  arg(5, [128][128], out)
        Reconstruct2(8, 128, (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockChol_out[0][0], (Cplfloat(*)[8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_ColumnUpdate_

        // Call to fusion_F_R_F4_Subtract
        fusion_F_R_F4_Subtract(8, (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4].S3.F_X_2_out[0][0][0][0], (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_Li

        // Call to fusion_F_R_F2_DataReplace
        fusion_F_R_F2_DataReplace(8, (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_Subtract_out3[1][1][0][0], (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4


        idxTime++;
    }
    #pragma omp critical
    {
    // Communication from Cluster.Shared to x86.External_DDR
    #pragma smecy communication src(Shared,STHORM,(threadprivate_i_F_0)%4,0) dst(External_DDR,Host)
    memcpy(&UG->SegHost.S1.S2.F_X__out[0][0], &UG->SegClusters[(threadprivate_i_F_0)%4].S3.F_Reconstruct_out3[0][0], 128*128*sizeof(Cplfloat));

    // Call to ApplyWeights
    #pragma smecy map(Host)  arg(4, [128], in)  arg(5, [128][128], in)  arg(6, [128], out)
    ApplyWeights(128, 128, 128, (Cplfloat(*))&UG->SegHost.S1.SteerVect_out[0], (Cplfloat(*)[128])&UG->SegHost.S1.S2.F_X__out[0][0], (Cplfloat(*))&UG->SegHost.S1.Weights_out[0

    }

}
// Call to TestWeights
#pragma smecy map(Host)  arg(3, [21][128], in)
TestWeights(21, 128, (Cplfloat(*)[128])&UG->SegHost.S1.Weights_out[0][0], "./ref/weights_128x400x200_10_ref.txt");
```

Fig. 7.8 SpearDE IR1 piece of code

Listing 7.2 SMECY MCAPI primitives generated by Par4All

```
void smecy_func_fusion_F_F2_fft_CF_6 (void)   { {
 mcapi_status_t SMECY_MCAPI_status;
 size_t P4A_received_size;
 Cplfloat *p4a_STHORM_1_0_fusion_F_F2_fft_CF_1_msg;
 mcapi_pktchan_recv (P4A_receive, (void **)&p4a_STHORM_1_0_fusion_F_F2_fft_CF_1_msg,
       &P4A_received_size, &SMECY_MCAPI_status);
 SMECY_MCAPI_check_status (SMECY_MCAPI_status,"accel_smecy_gencode.c", __func__, 345);
 Cplfloat *p4a_STHORM_1_0_fusion_F_F2_fft_CF_1 = p4a_STHORM_1_0_fusion_F_F2_fft_CF_1_msg;
 Cplfloat p4a_STHORM_1_0_fusion_F_F2_fft_CF_2[193 * 32];
 fusion_F_F2_fft_CF (p4a_STHORM_1_0_fusion_F_F2_fft_CF_1, p4a_STHORM_1_0_fusion_F_F2_fft_CF_2);
 mcapi_pktchan_send (P4A_transmit, p4a_STHORM_1_0_fusion_F_F2_fft_CF_2, 193 * 32 * sizeof (Cplfloat)
       , &SMECY_MCAPI_status);
 SMECY_MCAPI_check_status (SMECY_MCAPI_status,"accel_smecy_gencode.c", __func__, 348);
 mcapi_pktchan_release (p4a_STHORM_1_0_fusion_F_F2_fft_CF_1_msg, &SMECY_MCAPI_status);
 SMECY_MCAPI_check_status (SMECY_MCAPI_status,"accel_smecy_gencode.c", __func__, 349); } }
```

The final MCAPI primitives then run on the GEPOP Posix simulator in this case.

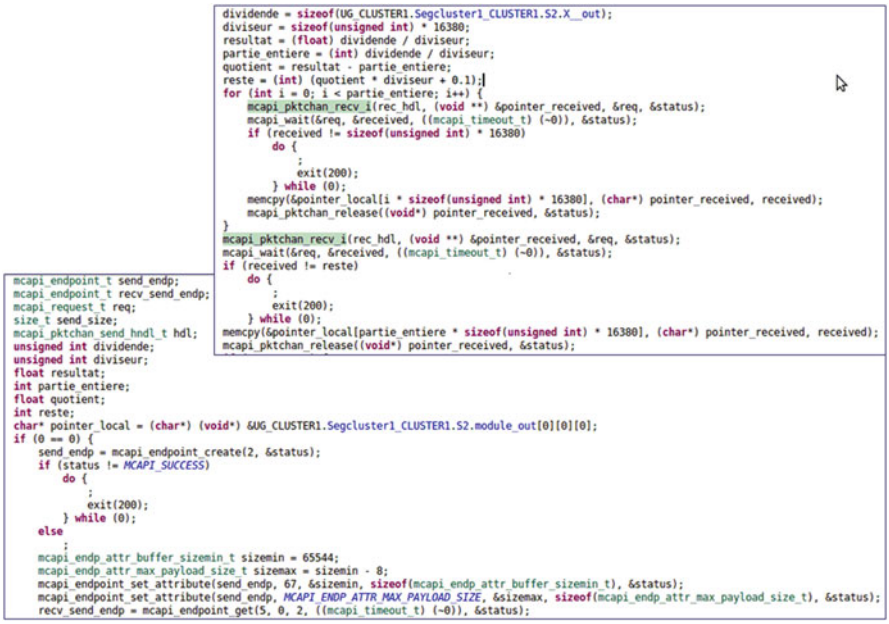

Fig. 7.9 SpearDE IR2 piece of code

7.2.3.2 SpearDE/IR2

Note that SpearDE is also able to generate the MCAPI IR2 from the same (graphical) parallelized application as seen in Fig. 7.7. Moreover SpearDE automatically manages the different communication protocols between communication ports of the host and the clusters (for instance no more than 16 ports per node are allowed by the MCAPI implementation). SpearDE also allows computation / communication overlapping (for latency optimization purpose).

Figure 7.9 shows the kind of MCAPI code that has been validated under the GEPOP Posix simulator.

7.3 Conclusion

This semiautomatic approach (through SME-C) brings some significant advantages:

- The user masters the parallelisation at high level.
- The parallelized application can be functionally tested at early stage since SME-C is executable (SME-C relies on pragmas that are close to standards like OpenMP).
- The user can rely on the design space exploration facilities for rapid prototyping purpose.
- A lot of parallelization information eases the back-end tool role (e.g., no need to analyze loops that are already declared as parallel).

Hence SME-C (generated by the front-end tool) provides the back-end tool with several hints that guide its work to generate an efficient low-level code on the target.

Acknowledgements This work also relies on the following Embedded Systems Lab's members: Teodora Petrisor (application modeling), Remi Barrere (tool enhancements, IR1 code generation), Paul Brelet (IR2 code generation) and Eric Lenormand (mapping). Claudia Cantini wishes to thank Prof. Marco Vanneschi and the Parallel Computing Laboratory of the Computer Science Department of the University of Pisa.

References

1. K. Cain, C. Torres, and R. Williams, "Rt-stap: Real-time space-time adaptive processing benchmark," 1997.
2. Wikipedia, "Cholesky decomposition," http://en.wikipedia.org/wiki/Choleskydecomposition, 2012. [Online]. Available: http://en.wikipedia.org/wiki/Choleskydecomposition
3. J. Choi, J. Dongarra, L. Ostrouchov, A. Petitet, D. Walker, and R. Whaley, "The design and implementation of the scalapack lu, qr and cholesky factorization routines," 1994.
4. E. Rothberg and R. Schreiber, "Improved load distribution in parallel sparse cholesky factorization," in *In Proceedings of Supercomputing '94*, 1994, pp. 783–792.
5. E. Rothberg and A. Gupta, "An efficient block-oriented approach to parallel sparse cholesky factorization," pp. 1413–1439, 1994.
6. E. Lenormand and G. Edelin, "An industrial perspective: a pragmatic high-end signal processing design environment at thales," in *In proceedings of the Workshop on Systems, Architectures, Modeling and Simulation SAMOS*, 2003, pp. 52–57.
7. SILKAN, http://www.par4all.org/, 2012. [Online]. Available: http://www.par4all.org/
8. MINES-ParisTech, "PIPS," http://pips4u.org, 1989–2009, open source, under GPLv3.

Chapter 8
Image Processing: Object Recognition

Marius Bozga, George Chasapis, Vassilios V. Dimakopoulos, and Aggelis Aggelis

8.1 The HMAX Algorithm

HMAX belongs to a class of feed-forward object recognition models inspired by the primate ventral visual pathway and defined by [1, 2]. HAI[1] serial (single threaded) implementation of the algorithm in pure C is based on the "hmin" implementation of HMAX algorithm in a C++-MATLAB environment by [3]. This translation from MATLAB-C++ code to pure C was accomplished with the help of the OpenCV library which replaced the MATLAB "functionality" of "hmin" implementation.

In this implementation we start by reading an image either from a file or from a network camera (Fig. 8.1, RI) and convert it to grayscale. This image is then scaled to 12 different scales ranging from 256×256 to 38×38 pixels (Fig. 8.1, SI). Up to this level all image reading and manipulation (conversion to grayscale and scaling) is performed with the help of OpenCV library.

[1]Hellenic Airspace Industries, Greece.

M. Bozga (✉)
UJF-Grenoble 1 / CNRS, VERIMAG UMR 5104, Grenoble, F-38041 France
e-mail: bozga@imag.fr

G. Chasapis
Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
e-mail: chasapis@eng.auth.gr

V.V. Dimakopoulos
Department of Computer Science and Engineering, University of Ioannina, Ioannina, GR-45110 Greece
e-mail: dimako@cs.uoi.gr

A. Aggelis
Hellenic Airspace Industries, Schimatari, Greece
e-mail: aaggel@haisat.net

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_8,
© Springer Science+Business Media New York 2014

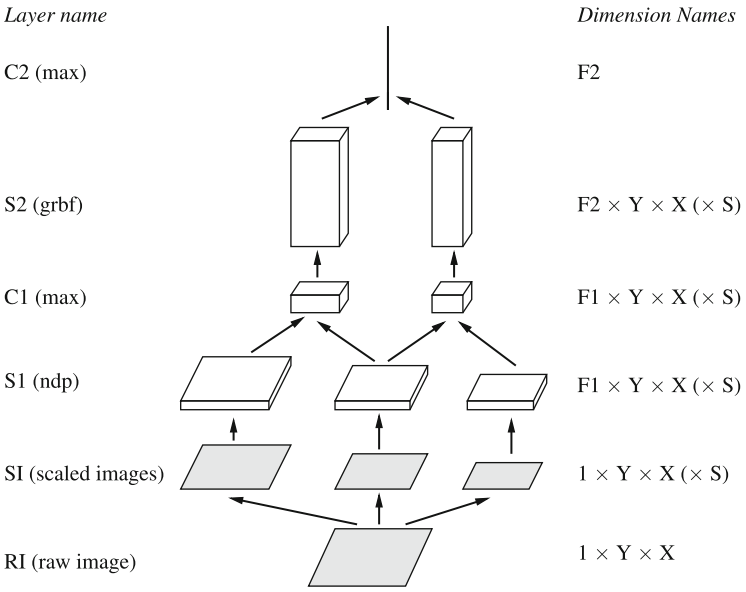

Fig. 8.1 HMAX hierarchy model

After SI, we successively compute image features "F" in the alternating S and C layers where:

- In Simple "S" layers the application of different local filters to the previous layer is computed.
- In Complex "C" layer invariance is increased by pooling neighboring S1 scales using the MAX operation. At the same time we subsample in both X and Y directions.

In the following paragraphs we present in more details layers S1 to C2 in our implementation.

S1 Layer: Gabor Filters

The S1 layer is the result of the application of 2D Gabor filters, of different orientations, on every possible pixel (since we cannot center Gabor filters over pixels near the edge of an image) on the scaled images (ranging from 256×256 to 38×38 pixels) of SI layer. In our specific model we use 12 Gabor filters with orientations "equally separated" to cover the whole 360°. 2D Gabor filters are 11×11 in dimension. Images have only one feature at each grid point which is the image intensity.

C1 Layer: Local Invariance

In this layer we pool nearby S1 units (of the same orientation) to create position and scale invariance over larger local regions. We also subsample S1 to reduce the number of units. S1 at its finer scale is $[246 \times 246 \times 4]$. For each orientation we compute a local maximum (MAX filter) over (X, Y) and adjacent scales. We also subsample by a factor of 5 in both X and Y. This results in S2 layer which at its finest scale is $[47 \times 47 \times 4]$.

S2 Layer: Intermediate Feature

In this layer we compute the responses R of C1 layer at every position and scale to a number of $D = 4075$ (dictionary) prototype patches $[4 \times 4 \times 4]$ (separately for each scale). This dictionary of patches is created by randomly sampling the C1 layer of training images. The resulting S2 layer at its finest scale is $[44 \times 44 \times 4075]$.

C2 Layer: Global Invariance

In this final layer a D-dimensional vector is created where each element is the maximum response in every position and every scale to the corresponding prototype patch. The result in our implementation is a 4075-D feature vector that can be used with a classifier of our choice. We have to note here that in this resulting vector all position and scale information is lost and we are left with a collection of features.

8.2 DOL/BIP-Based Parallelization

This section summarizes the experiments carried out on the HMAX algorithm using the DOL/BIP design flow. A preliminary version of these results has been presented in [4]. The DOL/BIP flow is a tool-supported flow for systematic and component-based construction of mixed software/hardware system models. System models are intended to represent, in an operational way, the set of timed executions of parallel application software statically mapped on a multiprocessor platform. As such, system models are used both for performance analysis using simulation-based techniques and for code generation on specific platforms.

The DOL/BIP design flow is rigorous and automated and allows fine-grain analysis of final hardware/software system dynamics. It is rigorous because it is based on formal models described in the BIP (*behavior interaction priority*) framework of [3], with precise semantics that can be analyzed by using formal techniques. A system model in BIP is derived by progressively integrating constraints induced on application software by the underlying hardware platform. The system construction method has been introduced in [5]. At user level, the application

software and the abstract model of the platform are specified using the DOL (*distributed operation layer*) framework defined by [5]. In contrast to ad hoc modeling approaches, the system model is obtained, in a compositional and incremental manner, from BIP models of the application software and, respectively, the hardware architecture, by application of (automated) source-to-source transformations that are proven correct-by-construction. The system model describes the behavior of the mixed hardware/software system and can be simulated and formally verified using the BIP toolset. Moreover, it can be used as a basis for automatic code generation for the target platform.

8.2.1 HMAX System Level Modeling in BIP

We briefly recall hereafter the construction of a mixed software/hardware system model introduced in [5] and illustrate it using fragments of the HMAX algorithm. The method takes as inputs representations of the application software, the hardware platform, and the mapping expressed in DOL. The output is the system model in BIP. The construction breaks down into several well-identified translation and model transformation phases operating on DOL and BIP models.

DOL is a framework introduced in [5] and devoted to the specification and analysis of mixed hardware systems. DOL provides languages for the representation of particular classes of applications software, multiprocessor architectures and their mappings. In DOL, application software is defined using a variant of KPN (*Kahn process network*) model. It consists of a set of deterministic, sequential processes (in C) communicating asynchronously through FIFO channels. The hardware architecture is described as interconnections of computational and communication resources such as processors, buses, and memories. The mapping associates application software components to resources of the hardware architecture, that is, processes to processors and FIFO channels to memories.

Figure 8.2 presents the process network model constructed from the S1/C1 layers of the HMAX models algorithm. It contains processes Splitter, GFilter1–GFilter12, MaxFilter1–MaxFilter11, and Joiner. The Splitter builds the 12 scales of the input image and dispatches them to Filters. Each GFilter1–GFilter12 implements a 2D-Gabor filter with different orientation. Their results are then sent, feature by feature, to MaxFilters. Each MaxFilter convolves outputs produced by two adjacent GFilters. The results are finally gathered by the Joiner. For the scope of this paper, we target a simplified, preliminary version of the STHORM platform. This version consists of a mono-cluster version of the STHORM fabric and an Encore engine featuring 16 PEs.

BIP is a formal framework introduced in [4, 5] for building complex systems by coordinating the behavior of a set of atomic components. Behavior is defined as automata or Petri nets extended with data and functions described in C/C++. The description of coordination between components is layered. The first layer describes the interactions between components. The second layer describes dynamic priorities

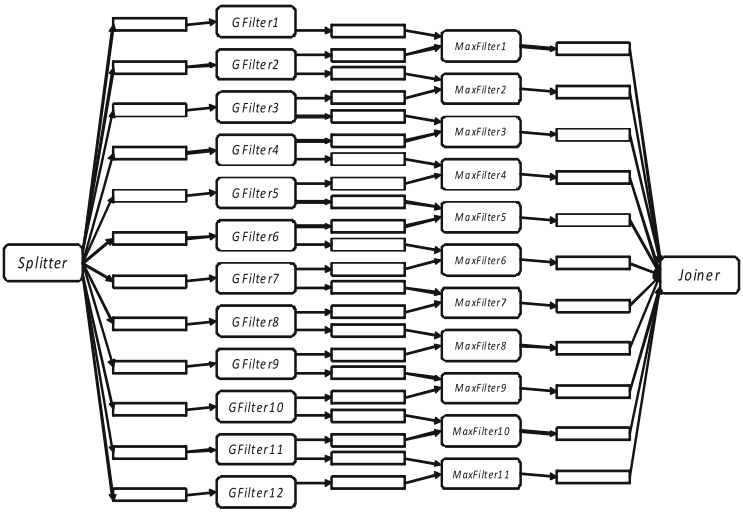

Fig. 8.2 KPN model of the HMAX S1-C1 layers in DOL

between interactions and is used to express scheduling policies. BIP has clean operational semantics that describe the behavior of a composite component as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model (transition systems) and its implementation. The BIP toolset includes a rich set of tools for modeling, execution, analysis (both static and on the fly) and transformations of BIP models. It provides a dedicated programming language for describing BIP models. The front-end tools allow editing and parsing of BIP programs, followed by code generation (in C/C++). The code can be used either for execution or for performance analysis using back end simulation tools.

As an example, Fig. 8.3 presents the model of a 2D-Gabor filter as an atomic component in BIP. This component consists of six control locations (START, S1,..., S4, DETACH) and two ports, DOL_read and DOL_write. NDPF_state, size, and address are local data (variables) of the component. The variables address and size are associated with the ports. The transitions are either internal transitions (internal_step) where local computation and updates are made, or port interactions, where the component exchanges data and synchronizes with the other BIP components.

The system model embodies the hardware constraints into the software model according to the mapping. The construction of the system model in BIP is obtained by a sequence of translations and transformations of the DOL representations, as follows:

1. Automatic translation of the application software in DOL into a BIP model. The translation is structural: processes and FIFO channels in DOL are translated into atomic components in BIP; connections in DOL are translated into connectors in BIP.

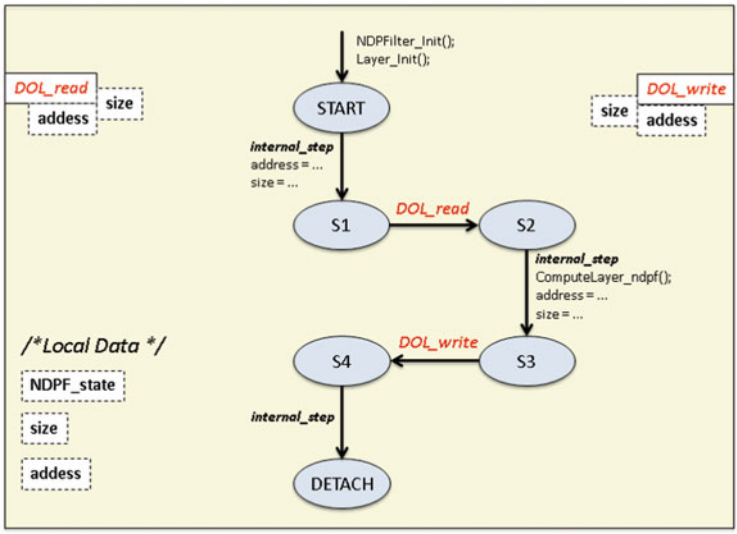

Fig. 8.3 BIP model of a 2D-Gabor filter component

2. Automatic translation of the hardware architecture model in DOL into a BIP model. The translation is also structural: hardware resources (processor, memory, bus, etc.) are translated into BIP components; hardware interconnections are translated into connectors in BIP.
3. Construction of an initial, abstract system model using source-to-source transformation of the previous two models and composition according to the mapping.
4. Refinement of the previous system model by including specific timing information about the execution of the software on the platform.

All the transformations above preserve functional properties of the application software model. Moreover, the system model includes specific timing constraints for execution of the application software on the hardware platform. These constraints are obtained by cross compiling the application model into executable code for the target and measuring the execution time of the elementary blocks of code (e.g., BIP transitions). The timing information is integrated in the system model through the source-to-source transformation done in last refinement step (number 4 in the list above).

For experiments, we restrict ourselves to the S1 layer of the HMAX models algorithm. The process network in DOL consists of 14 processes and 24 FIFO channels. This DOL model is about 700 lines of XML (defining the process network structure) and 1,500 lines of C (defining the process behavior). The software model in BIP is constructed automatically from the DOL model. It consists of 38 atomic components interconnected using 48 connectors. The BIP software model is about 2,000 lines of BIP code. The system model obtained by deploying the S1 layer on

a single STHORM cluster consists of 125 atomic components interconnected using about 1,500 connectors. The total BIP description totalizes about 13,000 lines of BIP code. This description is compiled into about 50,000 lines of C++ code, used for simulation and performance analysis, as explained below.

8.2.2 Performance Analysis on the System Model

The system model captures, besides the pure functionality of the application software, all the nonfunctional constraints induced on it by the target platform. The system model can therefore be used to analyze nonfunctional properties such as contention for buses and memory accesses, transfer latencies, and contention for processors. In the proposed design flow, these properties are evaluated by simulation of the system model extended with observers. Observers are regular BIP components that sense the state of the system model and collect pertinent information with respect to the properties of interest, i.e., the delay for particular data transfers and the blocking time on buses. Actually, we provide a collection of predefined observers allowing to monitor and record specific information for most common nonfunctional properties.

Simulation is performed by using the native BIP simulation tool. The BIP system model extended with observers is used to produce simulation code that runs on top of the BIP engine, that is, the middleware for execution/simulation of BIP models. The outcome of the simulation with the BIP engine is twofold. First, the information recorded by observers can be used as such to gain insight about the properties of interest. Second, with some cautions, the same information can be used to build much simpler, abstract stochastic models. These models can be further used to compute probabilistic guarantees on properties by using statistical model checking. This two-phase approach combining simulation and statistical model checking has been successfully experimented in a different context [6]. Moreover, it is fully scalable and allows (at least partially) overcoming the drawbacks related to simulation-based approaches, that is, the long simulation times and the lack of confidence in the results obtained.

For example, the results obtained on the HMAX application are presented in Fig. 8.4. The execution time of 2D-Gabor filters on STHORM PEs ranges from 220,106 to 068,106 cycles, depending on the size of the input image (ranging from 100×100 to 15×15 pixels). By using these values in the system model, the total execution time of the S1 layer is estimated as 225,106 cycles. This overall execution time is negatively impacted by the long access time (i.e., about 100 cycles) to the L3 memory (where all FIFOs are mapped) as well as by the bus contention. A slightly better result is obtained if the FIFOs are all mapped into the TCDM memory. In this case, the memory access time is about 1 cycle and there is no more contention. The total execution time reduces to about 220,106 cycles. However, such a mapping is not feasible due to memory size constraints, that is, FIFOs cannot fit all simultaneously within the TCDM memory.

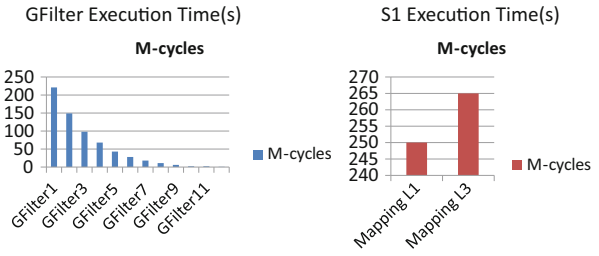

Fig. 8.4 Performance results estimated on the system model

8.2.3 Implementation and Experimental Results

We are developing an infrastructure for generating code from the BIP system models. We seek for portability and therefore, the generated code targets a particular runtime that can be eventually deployed and run on different platforms, including STHORM.

The runtime provides generic API for thread management, memory allocation, communication, and synchronization. The generated code is not bound to any particular platform and consists of the functional code and the glue code. The functional code implements the application tasks, that is, processes in the original DOL/BIP models. For each task, a C file is generated that contains the description of the data and a thread routine describing the behavior. The behavior is a sequential program consisting of computation statements and communication calls, that is, invocation of particular API provided by the runtime. The glue code implements the main routine that handles the allocation of threads to cores and the allocation of data to memories. Threads are created and allocated to processors according to the mapping description. Moreover, data allocation consists of allocation of the thread stacks and allocation of FIFO queues for communication. All these operations are realized by using the API provided by the runtime. The generated code is finally compiled by the native platform compiler. The code is linked with the runtime, hardware-dependent library, to produce the binary executable(s) for execution on the platform.

For our experiments, we have used the native programming layer (NPL), a common runtime implemented for both STHORM and MPARM platforms [7]. The generated code has been run on virtual platforms available in the STHORM SDK 2011.1, namely, GEPOP—the STHORM POSIX-based simulator—and the STHORM TLM simulator.

8.3 OpenCL-Based Parallelization

In Sect. 8.1 the HMAX algorithm was presented. It consists of four distinct layers of processing, S1, C1, S2, and C2, each one receiving the output of the previous one. The S1 layer involves filtering the input image, scaled into 12 different scales, using

a Gabor filter, for 12 different orientations. The filters dimensions are 11×11. The C1 layer is a MAX filter, selecting the maximum values for scale pyramids for a certain orientation. The S2 layer applies a GRBF filter between the C1 layer and a prototype set. Finally, the C2 layer is a global MAX filter of the S2 layer, resulting in a feature vector which corresponds to the initial input image.

Within the framework of the SMECY project, the intensive S1 part of the HMAX algorithm was programmed to run on a Gepop xp70 ISS multi-core STHORM platform which was presented earlier in this book. The parallel program was developed by using the OpenCL framework developed for the STHORM multi-core according to the OpenCL 1.1 specification from [8], and its use is explained in [9, 10]. We consider appropriate to present some basic concepts and terms extracted from the OpenCL framework specification which will allow the reader to comprehend the subsequent application program analysis and execution.

8.3.1 Basics of OpenCL

As it is stated in the OpenCL 1.1 specification:

> OpenCL is an open industry standard for programming a heterogeneous collection of CPUs, GPUs and other computing devices organized into a single platform. It is a framework for parallel programming and includes a language, API, libraries and a runtime to support software development.

The OpenCL framework uses a hierarchy of platform, memory, execution and programming model. The platform model consists of a host processor connected to a number of multicore computing devices, which are called “compute units". The application submits commands from the host to execute computations on the cores of the multicore compute unit. The syntax of these commands is defined in the OpenCL specification. OpenCL framework is offered in different versions, each being appropriate to support the runtime of a specific device, i.e., host or compute unit. The host version includes all the APIs that it can use to interact with the runtime, whereas the compute unit version is separate from the host runtime and compiler. Also, for each device a different language version may be supported, whereas in the case of multiple language versions the compiler selects the highest supported language version.

The execution model considers the execution of program to occur in two parts. The part that executes on one or more compute units and is called “kernel" and the part that executes on the host. The host program defines the context for the kernels and manages their execution. Instances of the kernel execute on the different cores of a compute unit. Each instance is called “work-item" and is identified by its position in an index space which is defined when a kernel is submitted for execution by the host. Work items are organized into “work groups". The work groups provide a more coarse-grained decomposition of the index space and are assigned IDs with the same dimensionality as the index space of the work items. Similarly, work

items are assigned local IDs within the work groups. In this way a single work item in a work group can be uniquely identified either by its global ID or by the combination of its local and work group IDs. The work items in a given work-group execute concurrently on the cores of a single compute unit. The index space is called “NDRange" and is an N dimensional index space where N is one, two, or three.

The work items have access to four distinct memory regions, that is, the global memory, the constant memory, the local memory, and private memory. In the global memory, all work groups and work items can read and write data; the constant memory is a region of the global memory in which the host places memory objects; in the local memory objects are placed that can be accessed by all the work items of a work group, and this memory is implemented as a memory dedicated to a compute device. The private memory is a memory region allocated to a work item. The application which runs on the host can use the OpenCL API to create memory objects in the global memory and to enqueue memory commands that operate on these objects.

The OpenCL execution model supports data-parallel and task-parallel programming models. The data-parallel model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object and the data are mapped to work items as the index space defines. In the task-parallel model a single instance of a kernel is executed independently of any index space.

Synchronization between work items in a single work group is achieved by the use of “work-group barriers". A prerequisite for a work item of a work group to continue is the prior execution of its work-group barrier.

8.3.2 Parallelizing HMAX Using OpenCL

The parallel version of the algorithm builds upon the sequential C version of the HMAX code, parallelizing the S1 layer of the algorithm, as it was mentioned above, leaving the rest of the sequential code unaltered. The S1 layer of the HMAX algorithm performs a convolution of the input image frames using a 2D-Gabor filter, for different scales and orientations. In this particular implementation 12 different image scales and 12 different Gabor rotations are used; thus we have 144 different image convolutions. The use of multiple scales of the input image and filtering using different orientations aims at achieving position and scale invariance by the pooling operation performed at the C1 layer.

The S1 parallelization was based on the data parallelism method which is the more efficient time-wise approach. The parallelization involves modifying the original sequential code, by transferring the functionality of the Gabor filter inside a kernel.

Three different versions of parallel code for the S1 layer have been written under the following assumptions:

- A different kernel object is placed in a queue for each scale defined in the host code. Hence the parallelization takes place in a more fine-grain level within an image scale, and consecutive computations for each image are executed sequentially.
- All the parallel implemented versions attempt to balance the computation load among the 16 available processing cores, by assigning as much as possible equal amounts of data for each core to process.
- The STHORM fabric is considered to have one cluster available, thus 16 cores which can execute parallel code. A possible extension would be to make the code more scalable in order to take advantage of a multi-cluster execution.
- Within the host code a 1-D NDRange is considered; thus the kernel which executes the Gabor filtering is called for 16 work items within one work group, assuming that each work item corresponds to one core processor. Since we have a mono-cluster execution, only one work group is defined.
- The Gabor filter mask itself is computed in the host code by GaborFilter_Init() function, which serves as a constructor for the pure C implementation of the filter.
- From a functional point of view, all three parallel versions have been correctly evaluated and the results from each case do not present any numerical deviations from the serial code.
- The host code computes the different scales of the input image from precomputed text files.
- Since the Gabor filter is a convolution, it involves computing an output pixel by moving the filter window over an area of the input image and executing the convolution at that particular position. The convolution computation itself for a particular output pixel takes place in one core processor, which is the core responsible for that very pixel.

8.3.3 First Version: Using Global L3 Memory

The first parallel version, which is the simpler one, employs only the global memory (L3 in the STHORM architecture) without utilizing the local memory (L1 in the STHORM architecture). The basic design principles are presented below:

- Within the kernel code, each work item is assigned a specific number of output image pixels, for which it computes the Gabor filter convolution. Figure 8.5 illustrates which pixels (painted in orange) are assigned to work item for processing. Image pixels for the other work items are assigned accordingly.
- No DMA memory transfers or explicit synchronizations whatsoever take place.
- Since each work item is responsible for transferring the needed data, no synchronization between work items is necessary.
- For the first parallel version a 3-D instead of a 1-D NDRange was actually employed, dividing the work items according to the vector $(f, x, y) = (2, 2, 4)$, where f refers to each Gabor rotation and $x - y$ refer to the respective image

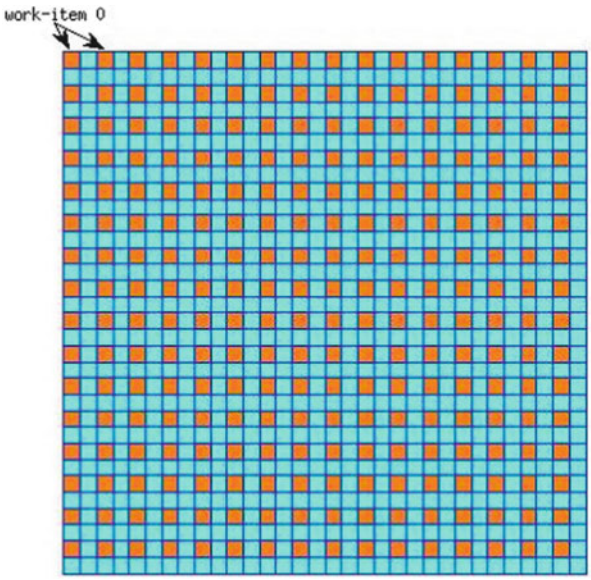

Fig. 8.5 Processing using only the L3 memory

dimension. Moreover, each work item is assigned pixels which are not adjacent to one another, thus are not contiguous in memory. Although only the L3 memory is utilized; this pixel assignment could have an impact on the performance, should any implicit memory transfers take place at a lower level.

8.3.4 Second Version: Liberal Approach

The second parallel version is based on the liberal DMA transfers approach, as mentioned in the STHORM programming guidelines for OpenCL document of [10]. In this version each work item defines private memory arrays and orders DMA copies for the input image into the private memory area. Computation is dispatched to each work item as follows:

- Each work item is responsible for computing a consecutive number of image lines. Care has been taken so that the lines of the image are equally distributed to work items, thus balancing computational load. To illustrate, in Fig. 8.6 each work item is assigned two image lines, where each image line consists of two DMA blocks.
- For every line assigned to it, each work item performs DMA transfers of the input data corresponding to that line into the L1 memory. The block size that is transferred with each DMA copy is defined inside the kernel code. Different block sizes were chosen in such a way that the impact on the execution performance to be visible.

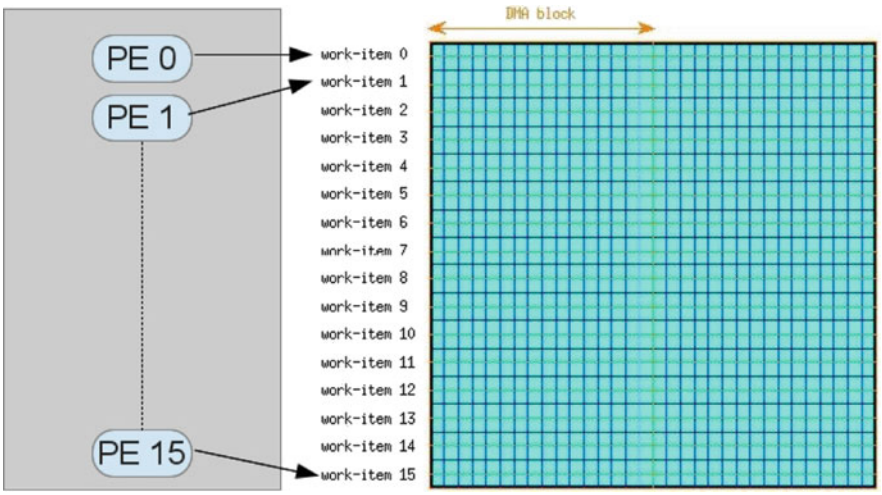

Fig. 8.6 DMA transferring with the private memory and PE mapping

- After the DMA copy is complete, the computation of convolutions for all the pixels in the private memory space takes place. The code was tested with the output lying on the global memory space only, as well as having the output first computed in the private memory, then being copied via DMA to the global memory L3.
- The above mentioned process has also been pipelined using double buffering, thus allowing an asynchronous copy to take place for the next data block while the work item computes the convolution for the current data block.
- Since the DMA transfers are more efficient when large chunks of contiguous memory are copied, the chunk size in this version should be as large as possible. However, due to the way the problem is partitioned, the largest possible chunk size for a single DMA transfer is limited to the width of the image, thus the code is not very flexible in terms of optimization.

8.3.5 Third Version: Collaborative Approach

The third parallel version is based on the collaborative DMA transfers, that is, the single work group issues a DMA transfer between the global and the local memory, where the local data will be used by all the work items. The basic strategies in this version of the code are as follows:

- As one can observe in Fig. 8.7, the computation is divided vertically between the work items, i.e., each work item computes the convolution for a number of consecutive image columns within the image block in the L1 memory.

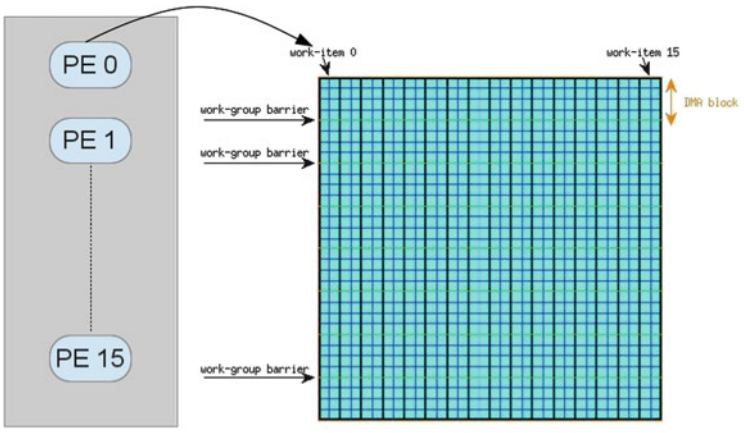

Fig. 8.7 DMA transferring with the local memory and PE mapping

- The DMA block for the input in this approach was chosen to be as coarse-grained as possible, in order to minimize the number of DMA transfers. As an illustration, in Fig. 8.7, four contiguous image lines are transferred via a single DMA transfer. Unlike the second parallel version, this version allows the programmer to issue more efficient DMA copies, having of course the disadvantage that synchronization is necessary before every new copy. Synchronization is achieved by issuing work-group barriers.
- The Gabor filter data is also copied into the L1 memory before computation.
- At the end of each iteration, synchronization is ensured via a local memory barrier.
- Given that the third approach requires significantly fewer DMA copies than the second approach, the simulation traces showed a reasonable increase in execution speed.

8.3.6 Experimental Results

The three aforementioned parallelization methods were executed using the Gepop xp70 ISS simulator on the fabric side, in order to obtain more accurate trace results that could reliably be compared to each other. The execution took place on an Intel Core2Duo E6750 machine with 3 GB of RAM available. The execution time for each parallel version was approximately 2.5 h.

The table below summarizes the kernel execution times for each implemented parallel version. The times are given in machine cycles.

As one can easily infer from Table 8.1, the third parallel version of the algorithm outperforms the other two in terms of execution speed, despite the fact

Table 8.1 Execution times for all the parallel versions

Parallel version	Total kernel time (cycles)	DMA copy time (cycles)	Waiting for events time (cycles)	Barrier synchronization time (cycles)
1	7, 516, 337, 152	0	0	0
2	3, 892, 169, 216	12, 576	503, 326	0
3	313, 308, 000	5, 497	652, 848	12, 602, 269

that a significant amount of time is spent during synchronizations. The superior performance of the collaborative approach lies to some extent on the fact that the number of DMA transfers is held relatively low, with each DMA transfer copying as much data as possible.

8.4 OpenMP-Based Parallelization

In this section we present the parallelization efforts for the HMAX algorithm using the OpenMP programming model. The presentation starts by visiting briefly the OpenMP implementation on the STHORM platform and then continues with the actual parallelization and its two variations.

8.4.1 OpenMP on STHORM

OpenMP [11] is the de facto model for programming shared-memory multicores. It is an intuitive, directive-based model whereby simple annotations added to a sequential C or Fortran program are enough to produce decent parallelism without significant effort, even by non-experienced programmers. As such, we believe OpenMP can form the basis for an attractive programming model beyond HPC, for multicore embedded accelerators.

In order to provide an implementation of the OpenMP model for the STHORM architecture we relied on source-to-source compilation. In particular we based our implementation on the OMPi compiler [12], a lightweight OpenMP V3.1 infrastructure for C, composed of a source-to-source compiler and a flexible runtime system. The compiler takes as input C code with OpenMP pragmas and outputs an intermediate multithreaded code augmented with calls to the runtime system. A native compiler is used to generate the final executable. We duly named our new compilation infrastructure *stOMPi*.

An initial observation is that the STHORM fabric is not a stand-alone device—it is rather a back-end system attached to a host, which could also be a multicore processor. There is a conceptual difficulty as to where and how the OpenMP programming model is to be applied: at the host side or at the fabric side? Our design decision was to be as general as possible, so as to have the greatest flexibility, and as

programmer friendly as possible, so as to let the programmer express parallelism in any way convenient. Consequently the goal of our compilation chain was to support OpenMP at both the host and the fabric levels.

From a programmer's point of view the application consists of two parts to be executed, respectively, by the host and the accelerator, which however are presented in a unified code. The accelerator part is a collection of C functions that are appropriately annotated. Function annotation occurs at the call site. The annotation model we follow is based on the SME-C representation [13] that has been proposed by the SMECY project consortium. In particular, a function call preceded by the following pragma:

```
#pragma smecy map(PE,n) [arg[[,]arg]...]
  <function call>
```

causes the called function ("kernel" in OpenCL terms) to be off-loaded to STHORM and executed (mapped) at PE with id equal to n. The optional "arg" clauses describe the size and direction of function arguments (input/output/both). In the off-loaded function code, OpenMP directives are allowed which dynamically spawn parallelism within the fabric. In addition, OpenMP in the rest of the user code is translated as parallelism to be generated at the host. Multiple host threads may off-load multiple functions for simultaneous execution on the fabric.

The stOMPi's compilation chain is composed of three phases. During the first phase, the compiler reads the source file and divides it into two new OpenMP files. The first file contains the code to be executed by the host processor and the second one has all the kernel functions to be off-loaded and executed on the MPSoC fabric. During the second phase the separated files are transformed into intermediate C code. The intermediate file for the host embeds calls to a standard OMPi runtime designed for the support of shared memory systems, which has been extended to provide the necessary primitives for communication with the STHORM fabric. The intermediate file for the fabric is augmented with calls to a new runtime that supports OpenMP execution within the accelerator. During the final phase, system back-end compilers are used to link the intermediate object files with the appropriate runtime libraries and create the two executables.

The parallelization techniques presented here target a single cluster because currently, the stOMPi tool chain provides full OpenMP support for a single STHORM cluster of 16 PEs while some limitations exist when utilizing more than one cluster. A detailed presentation of the design of stOMPi is given by [14].

8.4.2 Parallelizing HMAX Using OpenMP

In order to parallelize HMAX using OpenMP, we followed the same principles as presented in Sect. 8.3 for OpenCL. In particular, we focused our efforts exclusively on the S1 layer, which performs a convolution of the input image frames using a 2D-Gabor filter for 12 different scales and 12 different orientations,

for a total of 144 different image convolutions. The heart of S1 is function `GaborFilter_ComputeLayer` which performs the necessary computations.

In order to parallelize the S1 layer we started with the sequential version of the code and added OpenMP directives, since OpenMP is renowned for promoting *incremental parallelism*. The result is that the largest portion of the sequential program was left almost unchanged. It is notable that in the first parallelization effort, which we describe next, there was only one OpenMP line added to the sequential code, inside the `GaborFilter_ComputeLayer` function. This clearly accounts for high programmer productivity; of course, the obtained performance was not the best possible, but it nevertheless shows the practicality of the OpenMP model.

Two parallelized versions using OpenMP were designed and implemented, corresponding to the First version (Sect. 8.3.3) and Third version (Sect. 8.3.5) of the OpenCL parallelization. For both versions, the original (sequential) function `GaborFilter_ComputeLayer` was designated as the kernel to be off-loaded to the STHORM fabric for execution. This was implemented using the `smecy map` construct of the first-level intermediate representation (IR-1) defined by the SMECY consortium. In particular, the call to the function in question was simply preceded by a `#pragma smecy map (PE,0)` directive, followed by a number of `arg` clauses, describing the direction and size of the function arguments.

8.4.2.1 First Version: No DMA Transfers

In this version, similarly to the first version of the OpenCL-based parallelization, (Sect. 8.3.3), we followed a straightforward approach. All associated data (input, output, and filter mask matrices) are stored in L3 (or "shared") memory. Thus the corresponding data structures are equally accessible by the host and fabric. Computation-wise, an OpenMP thread computes a portion of the output pixels. We chose to distribute the load evenly to the 16 available threads in a way which is somewhat different to the method used in the corresponding OpenCL code. In particular, we started by letting each thread compute fully a number of output rows. To balance the load, we assigned the loop iterations that traverse the output rows using a `static` loop schedule, which divides the total number of iterations by the thread count. However, because the output sizes were mostly integers that are not divisible by 16 (the number of active OpenMP threads), in almost all cases a slight load imbalance was present, where some threads were forced to compute one more full row than the others. To alleviate this phenomenon, the `collapse` clause of OpenMP V3.1 was used in order to unify the row-scanning and column-scanning loops and balance the load evenly.

8.4.2.2 Second Version: Collaborative DMA Transfers

Corresponding to the third version of OpenCL parallelization, we followed a collaborative DMA approach. The `GaborFilter_ComputeLayer` kernel was modified in order to exploit the TCMD memory available locally to the cluster PEs.

Table 8.2 Simulator execution times for the OpenMP experiments

Parallelization approach	Cycles	Time (ns)	DMA transfer time (ns)
First version	19, 916, 432, 275	49, 791, 080, 689	0
Second version	1, 665, 365, 087	4, 163, 412, 717	13, 602, 204

In particular, PE 0 uses DMA to copy a Gabor filter mask from shared memory to TCDM. After that the same PE issues DMA requests for a block of input lines. The corresponding output block is then cooperatively computed by all 16 PEs. To achieve this a `parallel for` directive is used to parallelize the column-scanning loops. The computed output block is then copied back to shared memory by PE 0 using the DMA mechanism.

In order for the OpenMP programmer to use the DMA facilities of the device, stOMPi provides its own API for interfacing with the corresponding native platform primitives; the implemented functions act mostly as wrappers.

8.4.3 Experimental Results

Each of the two aforementioned parallelized versions of the HMAX application was compiled by the stOMPi compiler and linked with its runtime libraries. The first one utilized only a single OpenMP directive plus an IR-1 `smecy map` directive to off-load the kernel code to the fabric. The second one additionally needed some (limited) code restructuring and two calls to the stOMPi API to utilize the DMA facilities.

Both versions were executed in a server with a six-core AMD Phenom II X6 1055T processor with 4 GB of RAM, running Ubuntu Linux. The STHORM SDK version was 2012.2 which is able to simulate both the host and the fabric sides of the system. In Table 8.2 we summarize the execution times we obtained for the two OpenMP-based parallelized versions based on the statistics produced by the simulator. From the table, it should be clear that the second version results in significantly faster execution as compared to the first version, due to the extensive use of local memory instead of the quite slower shared (L3) memory. Another observation is that the execution times are higher than the ones obtained by the OpenCL-based parallelization. This is attributed to two factors:

1. The stOMPi OpenMP runtime is relatively "heavier" as compared to OpenCL. In particular, because of the need to store thread stacks, among others things, in TCDM, a portion of the TCDM is not available to the application, forcing a smaller DMA block size and thus more transfers from/to shared memory.
2. A more important issue, however, is that the SDK version utilized does not offer reliable timing measurements for all memory accesses. In particular, a portion

of the issued memory requests were categorized as accesses to "other" memory and were accounted as taking 30× the access time of TCDM, raising the total execution time by a significant amount.

Because future versions of the SDK will provide better timing facilities, we expect our timing results to drop sharply. Although we do not expect to surpass the performance of the highly optimized OpenCL implementation, we envision quite competitive timing. This fact, combined with the relatively low effort needed to parallelize the code, confirms our belief that OpenMP is a very appropriate model for multicore embedded accelerators.

References

1. M. Riesenhuber and T. Poggio, "Hierarchical Models of Object Recognition in Cortex," *Nature Neuroscience*, no. 2, pp. 1019–1025, 1999.
2. T. Serre, L. Wolf, and T. Poggio, "Object Recognition with Features Inspired by Visual Cortex," in *Proceedings of CVPR*. IEEE Computer Society, 2005, pp. 994–1000.
3. J. Mutch and D. G. Lowe, "Object class recognition and localization using sparse features with limited receptive fields," *International Journal of Computer Vision*, vol. 80, no. 1, pp. 45–57, 2008.
4. A. Basu, S. Bensalem, M. Bozga, J. Mottin, F. Pacull, A. Poulakidas, and A. Aggelis, "System Level Modeling, Analysis and Code Generation: Object Recognition Case Study," in *Proceedings of Embedded World Conference*, 2012.
5. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous Component-based Design Using the BIP Framework," *IEEE Software, Special Edition – Software Components beyond Programming – from Routines to Services*, vol. 28, no. 3, pp. 41–48, 2011.
6. A. Basu, S. Bensalem, M. Bozga, B. Caillaud, B. Delahaye, and A. Legay, "Statistical Abstraction and Model-Checking of Large Heterogeneous Systems," in *Proceedings of FMOODS/FORTE Conference*, ser. LNCS, vol. 6117. Springer, 2010, pp. 32–46.
7. L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *Journal of VLSI Signal Processing Systems*, vol. 41, pp. 169–182, 2005.
8. A. Munchi, *The OpenCL Specification, Version 1.1, Document Revision:44*, Khronos Group, 2011.
9. T. Lepley, *P2012 OPENCL SDK User Manual*, STMicroelectronics, 2012.
10. T. Lepley, *OpenCL Programming Guidelines for P2012*, STMicroelectronics, 2012.
11. OpenMP ARB, *OpenMP Application Program Interface V3.1*, July 2011.
12. V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP V.2.0," in *Proceedings of EWOMP Workshop*, 2003, pp. 5–11.
13. M. Torquati, M. Vanneschi, M. Amini, S. Guelton, R. Keryell, V. Lanore, F.-X. Pasquier, M. Barreteau, R. Barrère, C.-T. Petrisor, É. Lenormand, C. Cantini, and F. D. Stefani, "An innovative compilation tool-chain for embedded multi-core architectures," in *Proceedings of Embedded World Conference*, 2012.
14. S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying OpenMP on an embedded multicore accelerator," in *SAMOS XIII, 13th Int'l Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation*, Samos, Greece, July 2013.

Chapter 9
Video Processing: Foreground Recognition in the ASVP Platform

Petr Honzík, Roman Bartosiński, Martin Daněk, Leoš Kafka, Lukáš Kohout, and Jaroslav Sýkora

9.1 Introduction

Current video surveillance systems belong to two classes based on two technology concepts. The first and older concept is based on analog video systems that have been widely used up till now. The second more recent concept is based on the IP network technology, which enables much more variability in configuration and communication among all components connected in a network. IP-based video systems will slowly replace all analog video systems. In our approach we assume that in the future any network type will be available anywhere in the public and private space. All electronic appliances will be able to connect to each other over the network. Because of this reason in our design we need to consider autonomous equipment that can decide and connect to other devices in the network. In the future, as the number of monitored areas increases, there will be no chance to keep the current concept of video surveillance that relies on transferring complete image data from a big pool of cameras to one remote processing unit that processes all the data from the cameras as it would increase extremely the communication demands on the network, and its reliability is limited by the central processing node. Moreover, transferring all video data over a network brings in a big security risk. For this reason we need to process video data and make decisions locally and send only a minimum amount of data to the network to increase safety, processing capacity and dependability of the whole system. To sum it up, significant features of modern video surveillance systems are

P. Honzík (✉)
CIP plus, s.r.o., Milinska 130, Pribram 261 01, Czech Republic
e-mail: petr.honzik@cip.cz

R. Bartosiński • M. Daněk • L. Kafka • L. Kohout • J. Sýkora
UTIA AV CR, v.v.i., Pod Vodarenskou vezi 1143/4, Praha 8, 182 08, Czech Republic
e-mail: bartosr@centrum.cz; xdanek@email.cz; leos.kafka@centrum.cz; kohoutl@utia.cas.cz; sykora@utia.cas.cz

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_9,
© Springer Science+Business Media New York 2014

- Autonomous execution
- Local decision making
- Decentralization
- Increased privacy
- Cooperation with the outside world

Privacy is one of today's biggest problems, and most video surveillance system cannot meet strict rules dictated by privacy of personal spaces. Current video surveillance systems collect, store and send image data, which all increases the potential for illegal privacy intrusion. Because of this reason current video surveillance systems cannot be used to detect dangerous situations in many places like private rooms or social facilities, and there we have to count only on personal involvement of persons present there. If we are able to process video data locally and send out just detected events, we will be able to increase the use of video surveillance systems in these private spaces.

Modern video surveillance systems built to respect privacy issues must be based on *smart cameras*. A smart camera consists of an input device (mostly represented by a digital camera chip) and input video processing chain. These parts have crucial impact on the quality of outputs (events) generated by the system.

Figure 9.1 shows an example of an input video processing chain that we will consider in the remainder of the text. The input video processing chain consists of several processes that can be seen as three processing levels [1]. Generally, the amount of data in the chain decreases and the algorithm complexity increases with a switch to a higher processing level. Processes on the low and middle levels are usually where the data throughput bottleneck is. On the other hand, processes on the lower levels can be often parallelized.

The low-level processing usually contains simple operations that are repeated for each pixels (e.g. color transformations, filtering). As these processes do not need any additional memory to store contextual data or they need to store only several common parameters, they can be implemented as high-speed pipelined IP cores in an FPGA.

On the other hand, mid-level processes (e.g. object segmentation and tracking) require more complex and iterative or recursive algorithms with additional contextual data. In general, mid-level algorithms still operate on the whole video data. More sophisticated algorithms need higher amount of contextual history data than can be extracted from a single input frame.

The core block in the mid-level processing of video surveillance applications is *motion detection*. This process creates a mask of moving objects from a sequence of time-delayed input frames. The mask is usually computed using statistical algorithms. The noise in such a result is usually cleaned in the next operation; traditionally *morphological opening* is employed. Then connected pixels are identified as objects by a *labelling* algorithm. The output from this processing level is a mask of foreground objects and a table with parameters of individually identified objects.

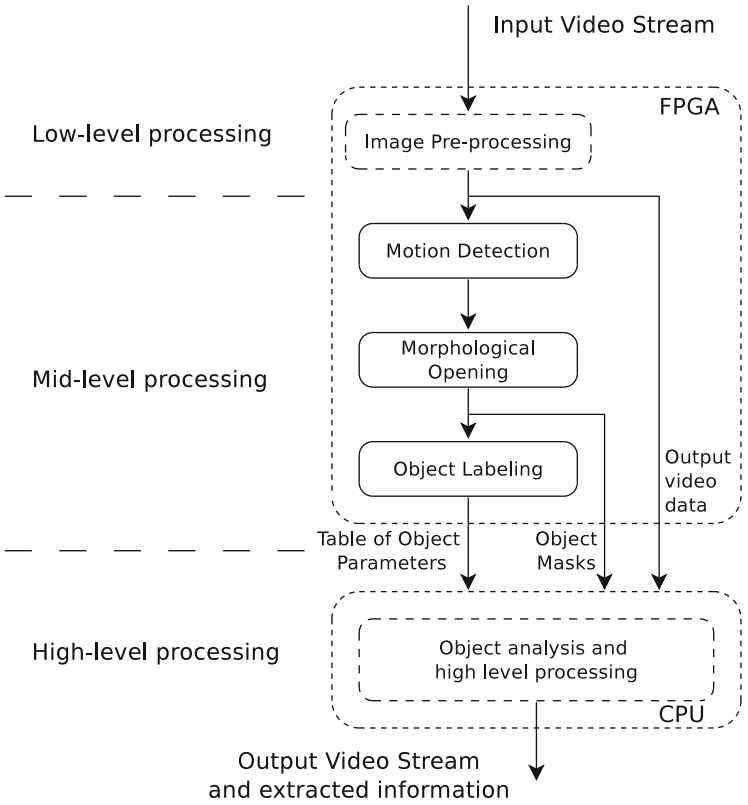

Fig. 9.1 Possible structure of an input video processing chain. Arrangement of component blocks in hardware resources used in our case

The high-level processes (e.g. object analysis and recognition) often involve complex sequential algorithms that usually execute on small regions determined in the middle level; these are best suited to be implemented in a CPU.

9.2 Platform

The implementation platform shown here is the application-specific vector processor (ASVP) described in [2, 3]. In traditional workflows based on direct implementation in hardware description languages hardware accelerators must be recompiled and re-synthesized each time to generate a new FPGA configuration (bitstream) that can be verified. The drawback of the traditional approach is the long synthesis time caused mainly by a slow place and route process of the low-level tools. This also limits the end user to minor changes of the implemented

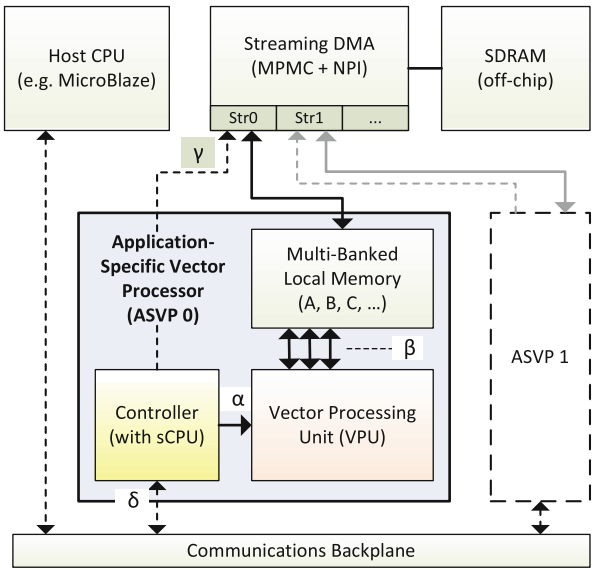

Fig. 9.2 A system-level organization of an ASVP-based core

algorithm such as tuning of coefficients. In our approach we abstract the custom accelerator into a specialized programmable architecture based on a network of programmable data-streaming computing nodes with local memory. To design for the platform, in the first step the high-level source code is analysed and domain features are extracted. Based on the required domain features new computing kernels are implemented in the computing nodes or the existing ones are modified. The architecture is programmable by firmware to the extent that the sequencing of basic processing kernels can be changed without resynthesizing the hardware; hence source code modifications that do not change the problem domain can be tested quickly for they require only fast firmware recompilation.

The system-level view is presented in Fig. 9.2. Similar to the streaming architectures (Stanford Imagine [7], IBM Cell [1]), the execution control is hierarchical:

1. *Task scheduling* is executing in the Host CPU (e.g. MicroBlaze). Optional inter-core synchronization is handled by the Communications Backplane (δ).
2. *Scheduling of the vector instructions* is realized in a simple scalar control processor (sCPU) embedded in each ASVP core. The sCPU forms and issues wide instruction words (α) to the vector processing unit (VPU).
3. *Data path multiplexing* and vector processing are realized autonomously in the VPU. The unit handles both the vector-linear and vector-reduction operations, as well as local memory banks access scheduling (β). The VPU contains data flow unit (DFU) which performs vector operations.

9.3 Application

9.3.1 Requirements

We have performed several tests to measure the speed of video processing required in real-world situations. The main parameter for video surveillance is the frame rate of the video processing block and video surveillance camera. The minimal frame rate of the video surveillance camera can be set according to the setting. Five feet per second is valid for common use in public spaces with people walking. In the case of moving cars or objects moving faster than pedestrians we need a higher frame rate. From our tests and experience it is desirable that in two subsequent frames the distance objects travel be 0.3 m for people and 0.6 m for cars. Table 9.1 shows the frame rate in three most common scenarios.

The next parameter is the minimal required resolution of the input video. The increasing demands on video quality imply the increase in the resolution of the input video stream. Currently cameras with the full HD resolution are often used as input of video stream. Hence we need to consider image resolutions starting at 640×480 pixels.

Another strong requirement is that the used algorithm must not exhibit faulty behaviour with varying external conditions such as time-varying lighting. The implemented algorithms should be adjustable by the user.

The main idea is that the application should respect privacy issues and provide image information only if a monitored event occurs.

To fulfil the stated requirements and in line with [1] we have selected the FPGA technology for implementing the low- and mid-level processes (shown in Fig. 9.1); the high-level analyses will be executed in a CPU.

As it is not efficient to design, implement and debug complex function cores directly on an FPGA (i.e. in a hardware description language), we have developed and analysed all algorithms in Matlab, and then we have implemented them in hardware. The next subsections analyse algorithms used in the processing blocks in our video chain.

9.3.2 Motion Detection

The motion detection process detects changes in an input video stream and returns mask of presently changed regions in the frame. This process is often

Table 9.1 Required frame rate versus object motion speed

Type of object	Speed (km/h)	Frame rate (fps)
People	5	5
Cars in a city	50	23
Cars on a highway	130	60

Table 9.2 The analysed algorithms for motion detection and memory required to store their contextual data normalized w.r.t the size of the input video frame

Algorithm	Normalized size of contextual data	Size for 640 × 480
Simple background subtraction	1	900 kB
Spatial temporal entropy image [4]	N+1/3 (for N quantization levels)	23.7 MB
Mixture of Gaussians [8]	8*K (for K Gaussian models)	37.5 MB
Modified mixture of Gaussians	5*K (for K Gaussian models)	23.4 MB
Bayes classifications [5]	3*N+4*M+3 (for N color vectors and M co-occurrence vectors)	247.5 MB

The last column shows the size of contextual data for the recommended settings and video resolution 640 × 480

called foreground/background segmentation, where static regions are marked as background and changes as foreground objects. There are many algorithms for motion detection, from simple background subtraction to complex algorithms which are based on statistical models of the background. In general, simpler algorithms are faster and need smaller amount of memory to store their contextual data. More robust and sophisticated algorithms are more computationally intensive and require higher amount of data memory. All algorithms use one or more parameters (mainly thresholds) that are adjustable and must be set by the user, but the quality of their values is crucial to get useful results. These parameters are commonly set experimentally.

We have analysed several algorithms for their computation complexity, data consumption and robustness. In the analysis we used the settings recommended in papers [4, 5, 8] where the algorithms have been described. The algorithms and the required memory size for their contextual data are listed in Table 9.2. The table also contains calculated sizes for the recommended parameters and color input video with resolution 640 × 480 pixels. Figure 9.3 demonstrates the basic quality of analysed algorithms; it shows the number of false-positive errors each algorithm produced in relation to the video resolution. The algorithms have been tested on artificial video sequences prepared in a scene modelling program.

Except simple background subtraction all tested algorithms require floating-point arithmetic.

Based on the analysis we have selected the algorithm based on the mixture of Gaussians (*MoG*) as a compromise between computational complexity, required memory for contextual data, used operations and the possibility to transform the algorithm to a vector form. The selected algorithm has been modified to decrease its memory requirements. The algorithm is based on [8] without shadow detection. This algorithm belongs to the class of *pixel motion detection* algorithms which consider all pixels as independent.

The algorithm is executed for each new frame from an input video in RGB or another three-component color space. The output of the algorithm is a mask of foreground objects in the frame.

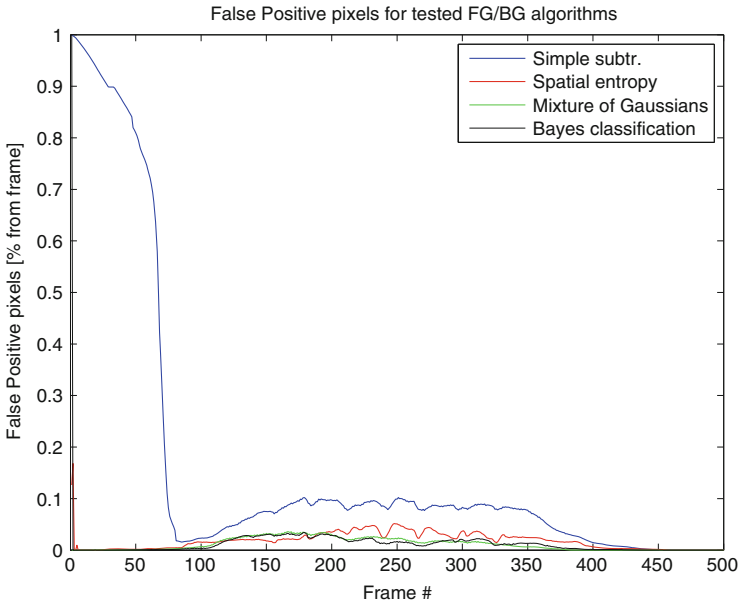

Fig. 9.3 False-positive error frequency for a testing video sequence

The decisions if pixels from the current frame represent foreground or background depend on statistical models and their mixture for each pixel. Each pixel is modeled by a mixture of K strongest Gaussians models of the background in the pixel ($K = 4$ in our implementation). Each Gaussian model defined by its mean value and variance represents one state (color) of a pixel. Each model also contains a parameter *weight* which represents how often the particular model classified the pixel as the background.

Four models are sufficient to describe basic and repeating changes of the scene background such as trees in air and moving escalators.

The algorithm tests if the current pixel value belongs to one of the already existing pixel models. It tests models sequentially from the strongest to the weakest one. If there is an appropriate model, the model is updated with the new value (the mean value and variance are updated and the weight of the model is increased). The model is updated with a given learning rate to perform progressive adaptation to new conditions. If none of the models describes the pixel, the weakest model is replaced with a new one which is built based on the current pixel value and with the default weight. After updating the models, they are reordered from the strongest to the weakest one, and their weights are normalized. Then the algorithm tests if the weight of the model (or mixture of stronger models) that describes the new pixel value is higher than the user's threshold; if so, the pixel is classified as background. Otherwise it is classified as a foreground object.

9.3.3 MoG Implementation

In its basic form the algorithm is suitable for sequential processing. For processing on the proposed ASVP architecture and with respect to the requirements it has been modified and transformed to a vector form. The main parts changed are conditional branching, reordering of the models, replacement of the operations divide and square root.

Each model in the original version is described by its mean value and variance in each color channel, weight and sort key. Thus each model is represented by eight single-precision floating-point (FP) numbers. In our modified version of the algorithm we reduced the model data to weight, mean value in each color channel and a common sum of variances, which is five FP numbers that equal to the reduction of the transferred contextual data by about 37.5%.

In the original version the operation *sqrt* was used to compute sort keys as a relation between the weight and variances of a model. Our modified algorithm uses only the weight to find the strongest and the weakest models. The modified version does not use the division to normalize weights; this is possible because the algorithm is recursive, and the weights of the models are updated in each step (with each new frame) by multiplying them with the forgetting factor that is always smaller than 1; thus the sum of weights for a given pixel cannot be greater than the number of models (four in our case).

The behaviour of both the original and the modified algorithms can be tuned with several parameters: background threshold, variance threshold, learning rate, initial weight, and initial variance.

The modifications of the algorithm have impact on the behaviour of the algorithm and its convergence, but the impact can be partially compensated by slightly adjusting parameters of the algorithm. The modified algorithm returns results with a slightly higher error constituent; it returns about 1% more false positive and about 1% fewer false-negative errors.

9.3.4 Morphological Opening

The second block in the mid-level processing (Fig. 9.1) performs morphological opening. It cleans the mask of foreground objects from the smallest objects which are mostly manifestations of noise or statistical errors. The opening algorithm is the dilation of the erosion of an input image. These operations are well known, and their description is included in many image processing textbooks (see, e.g. [6]).

9.3.5 Object Labelling

The input to most object labelling algorithms is a binary mask of objects. Some of the more complex algorithms use both the binary mask and the original colour

(or greyscale) image as their input. Each algorithm produces either a color mask where each separate object has a unique color, or it produces a table of objects with their description. The minimal description should contain a bounding box of each object (object position and its size) and its centre of mass. To generate a colour mask object colouring must be executed in two sequential steps. In the first step the algorithm marks objects with consecutive colours and produces a table of equivalent colours. In the second step the objects are re-mapped to the colour that described the particular object in the previous step. After that other characteristics can be computed for each object. Common labelling algorithms based on colouring the input image is processed by lines, pixel by pixel. The eight-neighbourhood is used in most cases while the four-neighbourhood is used rarely. If a pixel is neighbouring with another pixel that has already been assigned a colour, the pixel is coloured with the same colour, otherwise a new colour is assigned to the pixel. If the pixel has more pixels with different colours in its neighbourhood, the algorithm selects one colour and saves all the other colours to a temporary table of colours assigned to each object.

If we need only the basic characteristics (bounding box, centre of mass, volume), we can use a simplified algorithm that colours objects and computes their characteristics in just one step. All these characteristics can be computed and updated recursively; therefore the labelling algorithm can also compute the other characteristics. The algorithm is based on a static table of object characteristics, where one of the characteristics is the colour used in the algorithm, and objects are coloured with more colours saved in the table separately.

The output of the algorithm is a table that contains the bounding box, mass centre, object volume, and colour of each object.

9.4 Implementation and Results

The ASVP approach first constructs a programmable architecture customized for a given application, then employs software techniques to develop firmware that implements the algorithm.

In this methodology we need to describe the algorithm as a sequence of vector operations running in the computing nodes because the synthesized hardware must support all the required operations. Then the algorithm is written as a firmware with a sequence of set-up and computing operations in the VPU. The firmware also controls data transfers between the off-chip shared memory and local memories through the streaming DMA. Because the ASVP uses dual-port local memories, operation processing and data transfers from or to the shared memory can overlap. The data throughput can be maximized by optimizing this overlap. The speed of computation and data transfer minimization depends on the mapping of the data vectors to the accelerator local memory. The best performance of the accelerator is achieved for long data vectors that minimize data transfers between the local and shared memories.

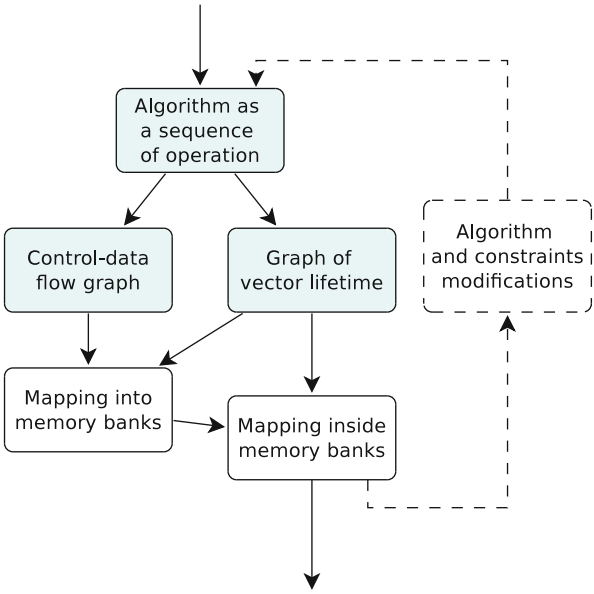

Fig. 9.4 Diagram of mapping process

The mapping problem can be solved through colouring of the data-dependency graph constructed from the sequence of operations and their variables. As the graph colouring is an NP-complete problem, the mapping is solved by heuristic algorithm.

If a solution is not found or if we want to optimize vector lengths, we can either insert the *VCOPY* operation to the operation sequence or use the time domain access scheduling in the VPU crossbar or try a different colouring.

Figure 9.4 depicts the mapping process. After all variables are mapped to the local memories, the data transfer schedule can be derived and a firmware program with operations and data transfers can be designed. In the future the firmware will be generated automatically from the input sequence of vector operations.

The whole video processing chain is developed and implemented on a Xilinx SP605 development board with a low-cost, low-power Xilinx Spartan 6 FPGA. The MicroBlaze soft-core processor is employed as the host CPU. Each algorithm is implemented in a separate ASVP. The entire system runs at 50 MHz except the DFUs that run at 100 MHz. All accelerators are synthesized with four memory banks with 1024×32-bit words.

9.4.1 Foreground/Background Segmentation

The architecture of the accelerator contains small local memories relative to the size of the input frame; therefore each frame must be processed in tiles with N pixels. In our implementation, each tile contains 50 pixels. The size of the tile is the maximal

Fig. 9.5 Mapping of variables to local memory A

possible in order to reduce data transfers. Mappings and lifetimes of variables in the implemented MoG algorithm are shown in Figs. 9.5–9.8. In the figures, the X-axis represents the time in the execution steps of the algorithm, and the vertical axis corresponds to the offset in memory. The second memory bank (Fig. 9.6) is occupied mainly by the pixel models of the background. Our version of the algorithm uses four models for each pixel. Each model is defined by its mean value for R, G, B color components, common variance (placed in the first memory bank) and model weight.

The input data and models for pixels in each tile must be transferred from the shared off-chip memory to the local memories. Then the accelerator executes the modified MoG algorithm for all pixels in the tile treated as vectors, then the updated models are saved back to the shared off-chip memory.

We have modified the algorithm to eliminate conditional branches, hardware-expensive operations (division, square root) and reordering. We also had to extend the basic ASVP platform with a number of instructions described below (also see Table 9.3). Branches are reimplemented so that the algorithm speculatively computes both possibilities (branch taken and not taken), and with a special operation *VSELECT* it selects the proper result from both branches according to the condition. Reordering of the models has been replaced with a selection of the strongest and the weakest models from all pixel models. The operations *INDEXMAX* and *INDEXMIN* have been added for this reason. These operations return the integer index of the element where the maximal/minimal value is located. The integer index can be used in subsequent operations that work only with a selected model.

Another example of application-specific vector instructions required by the MoG algorithm is the group of VCONVR / VCONVG / VCONVB instructions. These

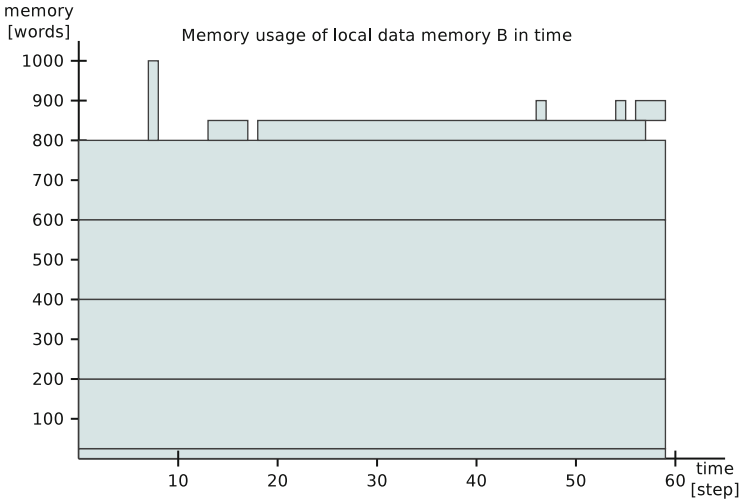

Fig. 9.6 Mapping of variables to local memory B

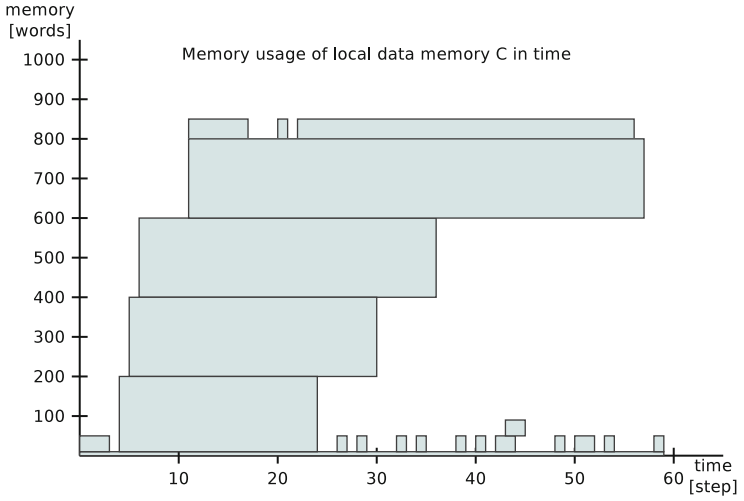

Fig. 9.7 Mapping of variables to local memory C

instructions take a 32-bit word that represents one pixel, extract a given 8-bit colour (R, G, or B), and convert the colour to a floating-point value. The VCMPLT (compare less than) operation compares two vectors element-wise and returns a vector of boolean values. The VSELECT operation is a vectorized conditional ternary operator as defined in the C language.

Given a set of operations and their high-level specifications in Table 9.3, the hardware implementation of the customized DFU can be generated. Currently this

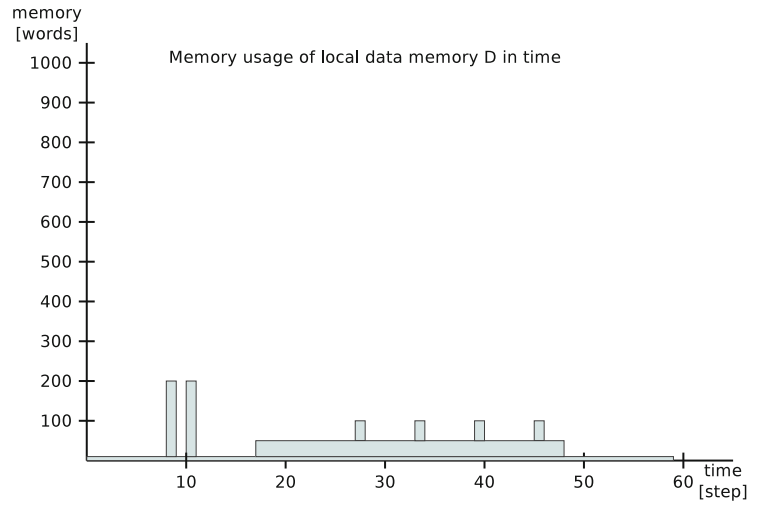

Fig. 9.8 Mapping of variables to local memory D

Table 9.3 Specific operations implemented in the DFU for motion detection

Operation	Definition
VMAX	$A_0 \leftarrow \Psi_{\text{Max}}(B_i)$
VMIN	$A_0 \leftarrow \Psi_{\text{Min}}(B_i)$
INDEXMAX	$A_0 \leftarrow Arg\{\Psi_{\text{Max}} B_i\}$
INDEXMIN	$A_0 \leftarrow Arg\{\Psi_{\text{Min}} B_i\}$
VCMPLT	$A_i \leftarrow (B_i < C_i)\ ?\ T : F$
VSELECT	$A_i \leftarrow (B_i \neq 0)\ ?\ C_i : D_i$
VGTE	$A_i \leftarrow (B_i < C_i)?C_i : B_i$
VBOR	$A_i = bitwiseOR(B_i, C_i)$
VBNOT	$A_i = bitwiseNOT(B_i)$
VCONVR	$A_i \leftarrow int2float((B_i >> 16)\&\ 0\text{xFF})$
VCONVG	$A_i \leftarrow int2float((B_i >> 8)\&\ 0\text{xFF})$
VCONVB	$A_i \leftarrow int2float(B_i\&\ 0\text{xFF})$

is done mostly manually in VHDL; however, it should be possible to synthesize the DFU automatically in a tool (this is our future work).

9.4.2 Morphological Opening

In our implementation, we want to be able to set the size of the structure element for morphological operations. Therefore we use a 3×3 square with the origin in its centre as a structure element. It has a feature that a sequence of N repetitions of

Table 9.4 Operations implemented in the DFU for morphological opening

Operation	Definition
VCOPY	$A_i \leftarrow B_i$
VAND3H	$A_i \leftarrow and(B_i, B_{i+1}); i = 0$
	$A_i \leftarrow and(B_{i-1}, B_i, B_{i+1}); i \in (0, N-1)$
	$A_i \leftarrow and(B_{i-1}, B_i); i = N-1$
VAND3V	$A_i \leftarrow and(B_i, C_i, D_i)$
VOR3H	$A_i \leftarrow or(B_i, B_{i+1}); i = 0$
	$A_i \leftarrow or(B_{i-1}, B_i, B_{i+1}); i \in (0, N-1)$
	$A_i \leftarrow or(B_{i-1}, B_i); i = N-1$
VOR3V	$A_i \leftarrow or(B_i, C_i, D_i)$

AND and OR are logical operations, i.e. the result of the AND operation is TRUE if all operands are TRUE

morphological operation with the basic structure element can be substituted for an operation with an N times bigger structuring element.

The opening operation consists of the erosion and the dilation operations. We work with binary images, that is, in binary morphology. This means we can replace the erosion operation with the logical *AND* among neighbouring elements, and similarly we can replace the dilation operation with the logical *OR* among neighbouring elements.

To maximize the ASVP utilization we have implemented four new operations in the VPU. These operations allow the VPU to treat each line in the image as one vector, each line will be read from and written to the shared memory only once. As the morphological operations described here are fixed-point by nature, they are executed in a separate computing node that implements just these new operations to save resources (compared to the floating-point computing nodes with operations described in the previous section).

Table 9.4 shows the instructions implemented in the DFU that are used in the opening function. Operations VAND3H and VOR3H get three consecutive elements from an input vector (line of image) and set the corresponding element in the output vector if all three elements are set (VAND3H) or one of them is set at least (VOR3H). On the contrary, operations VAND3V and VOR3V process three elements with the same index from three vectors (consecutive lines of the input image).

Figure 9.9 shows the firmware pseudocode for one part of the opening function. It performs the erosion operation on an input image in *Input Buffer* (resolution $W \times H$) with a full square structure element of size $1 + 2 * N$.

9.4.3 Results

The data path of the implementation is shown in Fig. 9.10. The labelling process is implemented in software executed in MicroBlaze in the current version.

Fig. 9.9 ASVP firmware pseudocode for the erosion operation used in opening function

```
SetReadingFrom(InputBuffer)
SetWritingTo(TmpBuffer1)
for (cn=0:N-1) {
 /* initiate */
 Line1 = VCOPY(cZero),W
 ReadLine(Out),W
 Line2 = VAND3H(Out),W
 ReadLine(Out),W
 Line3 = VAND3H(Out),W
 WriteLine(Line1),W
 /* process entire image */
 for (y=1:H) {
  Line1 = VCOPY(Line2),W
  Line2 = VCOPY(Line3),W
  ReadLine(Out),W
  Line3 = VAND3H(Out),W
  Out = VAND3V(Line1,Line2,Line3),W
  WriteLine(Out),W
 }
 Line3 = VCOPY(cZero),W
 WriteLine(Line3),W

 if (cn & 1) { /* switch buffers */
  SetReadingFrom(TmpBuffer2)
  SetWritingTo(TmpBuffer1)
 } else {
  SetReadingFrom(TmpBuffer1)
  SetWritingTo(TmpBuffer2)
 }
}
```

As mentioned above the video application has been implemented in the system on a chip with hardware compute cores on a Xilinx SP605 development board. The board contains power management chip accessible through PMBus. Power consumption has been measured by this chip on power rail that supplies power to the FPGA internal logic.

The average power consumption of the entire system on a chip with disabled accelerators is $Pavg = 433.2\,\text{mW}$. Table 9.5 contains performance data for the motion detection algorithm executed on one, two, and three floating-point compute cores for an input video stream with resolution of 640×480 pixels. It is shown that the time required to process one frame is a multiple of the number of used compute cores, but the consumed energy is the same. The initial requirements for the minimal frame rate 5 FPS can be reached with three cores running in parallel; each core computes one-third of each frame.

Table 9.6 shows average of consumed energy for processing one frame in opening function. The current implementation of opening function reaches frame rate 11.51 FPS. Consumed energy is 1.67 mJ for one frame and it is only one-tenth of energy consumed by motion detection operation.

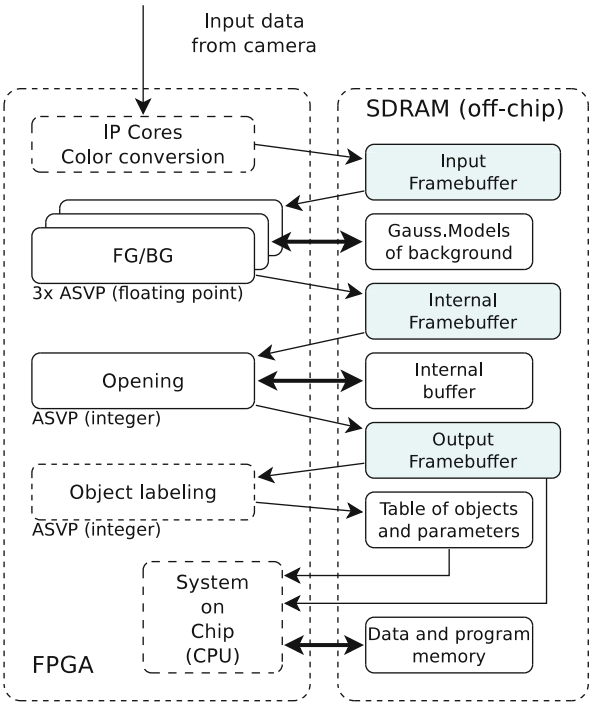

Fig. 9.10 The data path of the video chain

Table 9.5 Motion detection

	Number of used ASVP		
Parameter	1	2	3
$FPS[s^{-1}]$	2.07	4.14	6.21
P_{SoC} [mW]	460.26	487.58	509.68
P_{ASVP} [mW]	27.061	54.38	76.48
Computation of one frame			
t [s]	0.436	0.218	0.146
E [mJ]	11.81	11.87	11.13

Implementation parameters measured on Xilinx SP605

9.5 Summary

We have described an implementation of a video processing chain to be used in smart camera-based video surveillance applications. The implementation is based on the Xilinx SP605 development board. The implementation of the video processing chain is based on the ASVP platform, with the MicroBlaze running at 50 MHz and the computing nodes running at 100 MHz. Performance results for motion detection and morphological opening have been shown to meet the initial design requirements, and the power consumption data indicate that our implementation is suitable for low-power embedded devices.

Table 9.6 Morphological opening process

Parameter	Value
$FPS[s^{-1}]$	11.51
P_{SoC} [mW]	452.419
P_{ASVP} [mW]	19.219
Computation of one frame	
t [s]	0.0868
E [mJ]	1.67

Implementation parameters measured on Xilinx SP605

References

1. D. F. Real and F. Berry, "Smart Cameras: Technologies and Applications," in *Smart Cameras*, A. N. Belbachir, Ed. Springer US, 2010, pp. 35–50.
2. J. Sýkora, L. Kohout, R. Bartosiński, L. Kafka, M. Daněk, and P. Honzík, "The Architecture and the Technology Characterization of an FPGA-based Customizable Application-Specific Vector Processor," in *Proceedings of the 2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, ser. DDECS '12. IEEE, 2012, pp. 62–67.
3. J. Sýkora, L. Kohout, R. Bartosiński, L. Kafka, M. Daněk, and P. Honzík, "Reducing Instruction Issue Overheads in Application Specific Vector Processors," in *Proceedings of the 15th Euromicro Conference on Digital System Design*, ser. DSD '12. IEEE Computer Society, 2012, pp. 600–607.
4. G. Jing, C. E. Siong, and D. Rajan, "Foreground motion detection by difference-based spatial temporal entropy image," in *TENCON 2004. 2004 IEEE Region 10 Conference*, vol. A, 2004, pp. 379–382 Vol. 1. [Online]. Available: http://dx.doi.org/10.1109/TENCON.2004.1414436
5. L. Li, W. Huang, I. Y. H. Gu, and Q. Tian, "Foreground object detection from videos containing complex background," in *Proceedings of the eleventh ACM international conference on Multimedia*, ser. MULTIMEDIA '03. New York, NY, USA: ACM, 2003, pp. 2–10. [Online]. Available: http://doi.acm.org/10.1145/957013.957017
6. M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis, and Machine Vision*, 2nd ed. Chapman & Hall, 1998.
7. S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 31. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 3–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=290940.290946
8. P. Kaewtrakulpong and R. Bowden, "An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection," 2001.