Francky Catthoor · Twan Basten
Nikolaos Zompakis · Marc Geilen
Per Gunnar Kjeldsberg

System-Scenario-based Design Principles and Applications

Springer

System-Scenario-based Design Principles and Applications

Francky Catthoor • Twan Basten
Nikolaos Zompakis • Marc Geilen
Per Gunnar Kjeldsberg

System-Scenario-based Design Principles and Applications

Springer

Francky Catthoor
IMEC and KU Leuven
Leuven, Belgium

Nikolaos Zompakis
MicroLab-ECE-NTUA
Athens, Greece

Per Gunnar Kjeldsberg
Norwegian University of Science
and Technology
Trondheim, Norway

Twan Basten
Eindhoven University of Technology
and ESI, TNO
Eindhoven, The Netherlands

Marc Geilen
Eindhoven University of Technology
Eindhoven, The Netherlands

ISBN 978-3-030-20342-9 ISBN 978-3-030-20343-6 (eBook)
https://doi.org/10.1007/978-3-030-20343-6

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

In the past decades, embedded systems have become much more complex due to the introduction of a substantial amount of conditionally executed functionality in a single application and due to running of several applications or parts of applications concurrently. This substantially increases the dynamic nature of today's applications and systems, and it complicates dealing with their typically tight constraints in terms of, e.g., task deadlines, throughput, and battery lifetime. Also, optimizing for their cost functions like energy consumption and overall fabrication cost is much harder under such dynamic operating conditions. State-of-the-art design methods usually try to cope with these dynamic issues either by taking the corner cases, ending up in the true worst-case design paradigm, or by identifying several most used cases (use-case scenario paradigm) and dealing with them separately. Both of these approaches clearly reduce the complexity introduced by the dynamism, at the cost of adding large margins though.

In contrast, the material in this book encompasses a generic and systematic design-time/run-time methodology for handling the dynamic nature of modern embedded systems without adding such margins. It can be utilized in combination with most existing statically oriented realization methods to effectively deal with dynamism and to drastically increase their performance and efficiency. The presented method is based on the concept of system scenarios, which group system behaviors that are similar from a multidimensional cost perspective, such as resource requirements, delay, and energy consumption. This enables optimization per system scenario. It thus takes a more global cost-oriented system perspective, which is also the main distinction from the abovementioned use-case scenarios. One of the main aims with this book is to disseminate the know-how behind this concept more broadly.

Leuven, Belgium — Francky Catthoor
Eindhoven, The Netherlands — Twan Basten
Athens, Greece — Nikolaos Zompakis
Eindhoven, The Netherlands — Marc Geilen
Trondheim, Norway — Per Gunnar Kjeldsberg

Contents

Chapter 1
Introduction and Organization of Book Material

Francky Catthoor, Twan Basten, Nikolaos Zompakis, Marc C. W. Geilen, and Per Gunnar Kjeldsberg

1 Motivation and Context

In the past decades, embedded systems have become much more complex due to introducing a substantial amount of conditionally executed functionality in a single application, and due to running several applications or parts of applications concurrently. This substantially increases the dynamic nature of today's applications and systems, and it complicates dealing with their typically tight constraints in terms of, e.g., task deadlines, throughput, and battery lifetime. Also optimizing for their cost functions like energy consumption and overall fabrication cost is much harder under such dynamic operating conditions.

F. Catthoor (✉)
IMEC and KU Leuven, Leuven, Belgium
e-mail: catthoor@imec.be

T. Basten
Eindhoven University of Technology and ESI, TNO, Eindhoven, The Netherlands
e-mail: a.a.basten@tue.nl

N. Zompakis
MicroLab-ECE-NTUA, Athens, Greece
e-mail: nzompakis@microlab.ntua.gr

M. C. W. Geilen
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: m.c.w.geilen@tue.nl

P. G. Kjeldsberg
Norwegian University of Science and Technology, NTNU, Trondheim, Norway
e-mail: pgk@ntnu.no

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6_1

State-of-the-art design methods usually try to cope with these dynamic issues either by taking the corner cases, ending up in the true worst-case design paradigm; or by identifying several most used cases (use-case scenario paradigm) and dealing with them separately. Both of these approaches clearly reduce the complexity introduced by the dynamism, at the cost of adding large margins though. In contrast, the material in this book encompasses a generic and systematic design-time/run-time methodology for handling the dynamic nature of modern embedded systems without adding such margins. It can be utilized in combination with most existing statically oriented realization methods to effectively deal with dynamism and to drastically increase their performance and efficiency. The presented method is based on the concept of system scenarios, which group system behaviors that are similar from a multi-dimensional cost perspective, such as resource requirements, delay, energy consumption. This enables optimization per system scenario. It thus takes a more global cost-oriented system perspective, which is also the main distinction from the abovementioned use-case scenarios. One of the main aims with this book is to disseminate the know-how behind this concept more broadly.

At design-time, these system scenarios are identified and then individually optimized, typically in a parameterized way. Mechanisms for predicting and detecting the current scenario at run-time and for switching between scenarios at the most appropriate time slots are incorporated. This design trajectory is augmented with a run-time calibration mechanism, which allows the system to learn on-the-fly during its execution, and to adapt itself to the current input stimuli. This happens by extending the scenario set and/or changing the scenario definitions, and it affects both the prediction and switching mechanisms.

To show the generality of the system scenario methodology, we show how it has been effectively applied on many very different real-life design problems in the embedded system context. In all presented case studies substantial cost improvements (e.g., energy reductions) were obtained by exploiting system scenarios. These include several aspects of application to processor architecture mapping, e.g., based on the popular DVFS control knobs, scenario-aware data flow analysis, dependable system design, wireless baseband and protocol system design, biomedical systems like neuroprobes, manufacturing systems, and even dynamic memory management protocols, and photovoltaic system modeling and design.

2 Contributions of the Book and Target Audience

As mentioned, state-of-the-art design methods usually try to cope with the dynamic system issues in two basic ways: with worst-case corner cases or through use-case scenarios. Both of these approaches clearly alleviate part of the complexity problem, but they come at the cost of adding large safety margins in the design. In contrast, the system scenario approach discussed in this book avoids this.

The initial system scenario approach methodology was disclosed in [3]. The formalized methodology has been presented in [1] and then generalized further for

the control variable case in [2]. But since then, additional material has been provided and the approach has been extended beyond control variables, and beyond the initial application domains. Hence, we believe this book is a valuable way to provide a systematic and complete view of the approach.
For the reader, we offer:

- an effective solution to deal with dynamic system design;
- a broad survey of the state-of-the-art approaches and results in this domain;
- a large set of illustrative industrial-strength case studies covering a broad set of applications, substantiating the effectiveness and wide applicability of the approach.

3 Structure of the Book

The rest of this book is organized as follows: First, the overall flow is outlined and the different steps of the approach are summarized, including related work discussion in Chap. 2. The method is illustrated there also using control variable

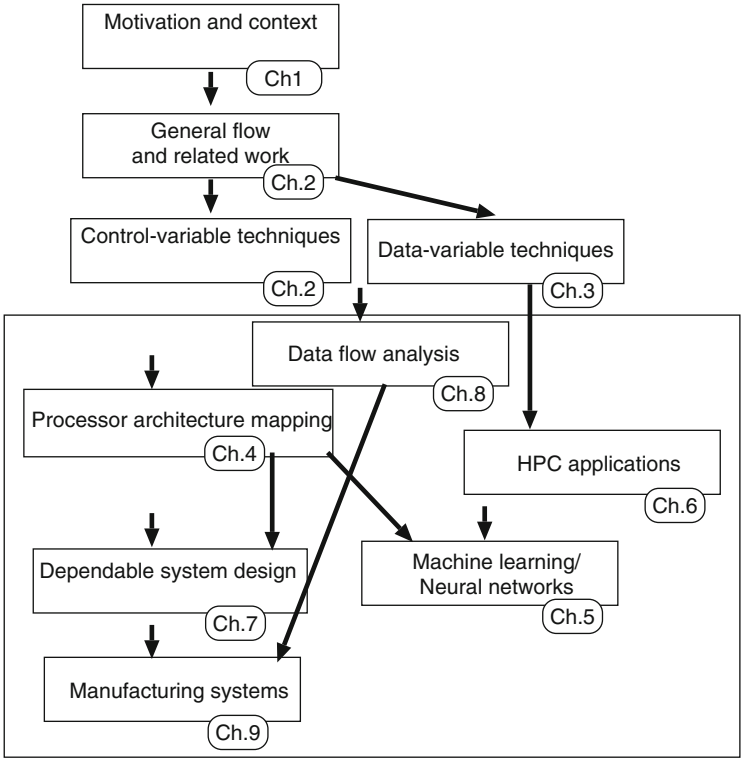


Fig. 1.1 Dependencies between book chapters

oriented techniques for system design optimization. Next, the extension to data variable oriented techniques is presented in Chap. 3. The following chapters provide case studies and demonstrations of the techniques in different application and system platform contexts. This includes application to processor architecture mapping in Chap. 4; to machine learning in Chap. 5; to HPC server systems in Chap. 6; to dependable system design in Chap. 7; to data flow analysis in Chap. 8; to manufacturing system design in Chap. 9. Remaining case studies are reviewed at the end of Chap. 2.

Figure 1.1 shows the main dependencies between these chapters. The initial chapters are required for a good understanding of the application in different domains. The control variable approach is mainly useful for the application domains mentioned in the left-hand side column of the big box, and the data variable variant is mainly needed in the right-hand side column.

4 Classification

Categories and Subject Descriptors [Computer Systems Organization]: Special-Purpose and Application-Based Systems|Real-time and embedded systems; [Software Engineering]: Design|Methodologies General Terms: Algorithms, Design, Performance.

Additional Key Words and Phrases Design methodology, dynamic nature, embedded systems, cost and energy reduction, real-time systems, scenario-based design.

Acknowledgements The material in this book is partly based on work in the context of several European and national research projects. In particular we want to acknowledge the support of: the EU Marie Curie project DACMA MEST-CT-2004-504767 and the Artist and HiPEAC Networks of Excellence; the EU FP7 project HARPA with grant number 612069; the EU Horizon 2020 project READEX with grant number 671657; the Flemish FWO (Fonds Wetenschappelijk Onderzoek Vlaanderen); the Netherlands Organisation for Scientific Research (NWO), through the FAME-612.064.101 project and through the Robust Cyber-Physical Systems (RCPS) program, projects 12694 and 12697; the ARTEMIS joint undertaking through the ALMARVI project (621439) and ITEA3 project 14014 ASSUME.

It has been a pleasure for us to work in this research domain and to cooperate with our project partners and our colleagues in the digital processor architecture design and embedded systems communities. Much of this work has been performed in tight cooperation with many university groups, mainly across Europe. In addition to learning many new things about system synthesis/compilation and related issues, we have also developed close connections with excellent people. Moreover, the pan-European aspect of this cooperation has allowed us to come in closer contact with research groups with a different background and "research culture," which has led to very enriching cross-fertilization. This is especially reflected in the many common publications. We want to especially acknowledge the valuable interactions and the excellent cooperation with the colleagues of the TU Eindhoven (The Netherlands); the El. Eng. Dep. of U. Ghent and the K. U. Leuven (Belgium); the ICCS department at the NTUAthens (Greece); the El. Eng. Dep. at the NTNU Trondheim (Norway).

We would like to use this opportunity to thank the many people who have directly helped us in realizing these results and who have provided technical and other contributions in the focus of this book, both at our institutes and at other locations. That includes all the students who helped us during the PhD. and M.S. thesis research.

References

1. S.V. Gheorghita, Dealing with dynamism in embedded system design: application scenarios, Ph.D. thesis, Eindhoven University of Technology, 2007. https://doi.org/10.6100/IR630369
2. S.V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, K. De Bosschere, System-scenario-based design of dynamic embedded systems, in *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14 (ACM New York, 2009)
3. P. Marchal, C. Wong, A. Prayati, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, H. De Man, Dynamic memory oriented transformations in the MPEG4 IM1-player on a low power platform, in *Proceedings of the First International Workshop on Power-Aware Computer Systems* (Cambridge MA, 2000), pp. 31–40

Chapter 2
System Scenario Methodology Flow

Francky Catthoor

1 Introduction and Context

The main focus of this book is on embedded system realization, even though the extension to high-performance computing (HPC) systems is partly addressed in Chap. 4. Embedded systems usually consist of processors that execute domain-specific applications. These systems are typically very software-intensive [37], having much of their functionality implemented in application and middleware software, which is running on one or multiple/many processors. As a result only the (too) high-performance kernels are left to more custom hardware implementation. Typical examples include cellular phones, wireless access points/gateways, multi-media applications like TV sets or MP3 players, automotive in-car systems, IoT sensor nodes, and body area networks (BANs).

All of these systems have complex workloads and stimuli sequences applied. Thus, these applications are full of dynamism, i.e., their execution costs (e.g., number of processor cycles, memory usage, energy) are strongly environment-dependent (e.g., input data, processor temperature). At the same time, many of these systems are running multimedia and/or communication applications and support multiple standards. And also for emerging domains like BAN systems, standards

Francky Catthoor (editor) based on material provided by Twan Basten, Valentin Gheorghita, Per Gunnar Kjeldsberg, Henk Corporaal, Elena Hammari, Yahya Yassin, Martin Palkovic, Juan Hamers, Arnaut Vandecappelle, Stelios Mamagkakis, Lieven Eeckhout, Frederik Vandeputte, and Koen De Bosschere.

F. Catthoor (✉)
IMEC and KU Leuven, Leuven, Belgium
e-mail: catthoor@imec.be

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6_2

are gradually set up to govern part of the functionality. So still, a large part of their behavior is predefined and it can be characterized quite well at design-time by profiling frameworks.

The rest of this chapter is organized as follows. Section 2 shows the distinction between more conventional use-case versus the system scenario concept which is our focus here. Next, Sect. 3 gives a motivating example for our research objectives, by showing how system scenario exploitation makes an H.264 video decoder significantly more energy efficient. In Sect. 4, the basic concepts and terminology behind the system scenario methodology are described. The overall system scenario methodology for embedded system design is introduced in Sect. 5. The detailed discussion of the main steps follows in Sects. 5.2, 5.3, 5.4, 5.5, and 5.6. In the rest of the book, multiple case studies are incorporated which describe a diversity of applications that fit within this methodology, illustrating its broad application potential. A brief summary of this wide range of applicability is provided in Sect. 6. However, in order to have an illustration of the main steps of the approach also in this chapter, a running case study based on control variable system scenarios is integrated here already in the above sections describing the main steps. The extension to multi-processor platforms and multi-tasking is briefly covered in Sect. 7. The most closely related work is presented in Sect. 8, and the conclusions and our future plans are detailed in Sect. 9.

2 Use-Case Versus System Scenario Concept

Scenario-based design [6] has been used for some time already in both hardware [38, 52] and software design [14] of embedded systems. In both of these cases, scenarios concretely describe the use of a future system, in an early phase of the development process. They appear like narrative descriptions of envisioned usage episodes, or unified modeling language (UML) use-case diagrams [23] which enumerate, from a functional and timing point of view, all possible user actions and the system reactions that are required to meet a proposed system function. In this book, this class of scenarios will be called use-case scenarios. They focus on the application functional and timing behaviors and on their interaction with the users and environment. In contrast to the objectives envisioned here, they abstract away or ignore the resources required by a system to meet its specifications and boundary constraints. Hence, these use-case scenarios are mostly usable as an input for design approaches centered around the application context.

In this chapter, we concentrate on a different and complementary type of scenarios, which we call system scenarios. These are derived from the combination of the behavior of the application and its mapping on the system platform. These scenarios are used to reduce the system cost by exploiting information about what can happen at run-time to make better design decisions at design-time, and to exploit the time-varying behavior at run-time. While use-case scenarios classify the application's behavior based on the different ways it can be used, system scenarios

instead classify the behavior based on the multi-dimensional cost trade-offs during the system mapping and implementation trajectory.

By optimizing the system per scenario and by making sure that the actual system scenario can be predicted at run-time, a system setting can be derived per scenario to optimally exploit the system scenario knowledge. To motivate the system scenario usage in embedded system design, we start from the different run-time situations (RTSs) in which a system may run. An RTS is a bounded part of the system execution that can be treated as an integral unit. Each RTS has an associated mapping cost, which usually consists of one or several primary costs, like quality and resource usage (e.g., number of processor cycles, memory size). The system execution is a sequence of RTSs, and the current RTS is known only at the moment it occurs. However, at run-time, using various system parameters, so-called RTS parameters, it can be predicted in advance in which RTS the system will run next for a non-zero future time window. If the information about all possible RTSs in which a system may run is known at design-time, and the RTSs are considered in different steps of the embedded system design, a better optimized (e.g., faster or more energy efficient) system can be built because specific and aggressive design decisions can be made for each RTS. These intermediate per-RTS optimizations lead to a smaller, cheaper, and more energy-efficient system that can deliver the required quality. In general, any combination of N cost dimensions may be targeted. However, the number of cost dimensions and all possible values of the considered RTS parameters may lead to an exponential growth in the number of RTSs.

This will degenerate to a long and overly complicated design process that does not deliver the optimal system at all. Moreover, the run-time overhead of detecting all these different RTSs will be too high compared to the expected gain over their (quite) short time windows. To avoid this situation, in our work, the RTSs are classified and clustered from an N-dimensional cost perspective, into system scenarios, such that the cost trade-off combinations within a scenario are always fairly similar, the RTS parameter values allow an accurate prediction, and a system setting can be defined that allows to exploit the scenario knowledge and optimizations. This chapter presents a systematic way of detecting and exploiting both at design-time and run-time the system scenarios of a given system setup. Generic solutions to the various steps in the methodology are provided whenever possible. The method combines design-time analyses and optimizations with information collected at run-time about the environment in which the system is operating and the inputs being received.

Figure 2.1 depicts a design trajectory using use-case and system scenarios. It starts from a product idea, for which the stake-holders manually define the product's functionality as use-case scenarios. The stakeholders are persons, entities, or organizations who have a direct stake in the final system; they can be owners, regulators, developers, users, or maintainers of the system. These scenarios characterize the system from a user perspective and are used as an input to the design of an embedded system. The latter will typically include both software and platform

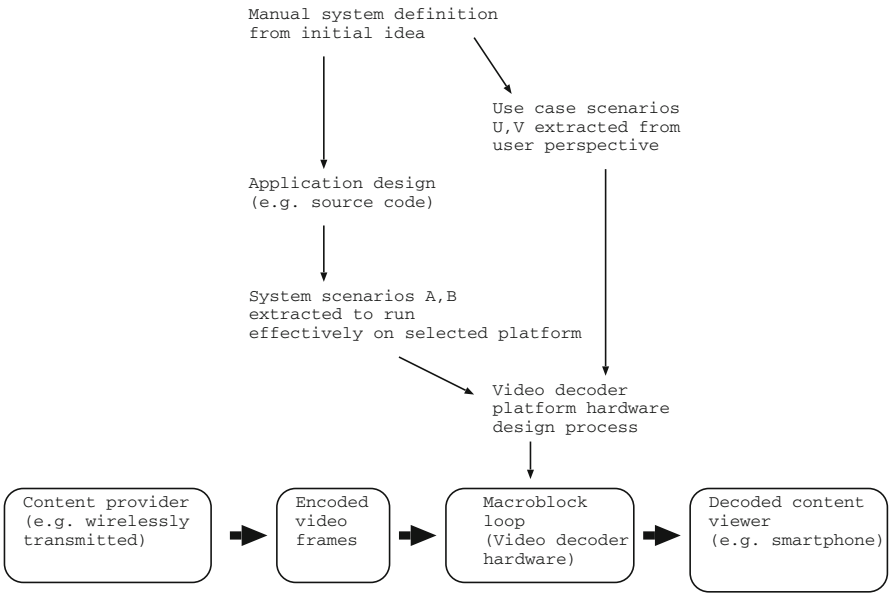


Fig. 2.1 Use-case vs system-scenario-based design flows for embedded systems

hardware components. In order to optimize the design of the system, the detection and usage of system scenarios augments this trajectory, as shown in the bottom gray box in the figure. The run-time behavior of the system is classified using the proposed methodology into several system scenarios, with similar cost trade-offs within each individual scenario. For each specific scenario, more customized and aggressive design decisions can be made than what is feasible for the global functional behavior. The sets of use-case scenarios and system scenarios are not necessarily fully disjoint, and it is possible that one or more use-case scenarios correspond to one system scenario. But still, usually they are not overlapping and it is more likely that a given use-case scenario is split into several system scenarios, or even that many system scenarios intersect several use-case scenarios.

3 Motivating Example

Figure 2.2 shows a typical system to which the system scenario design methodology is applicable. In this system, a content provider sends H.264 encoded sequences of video frames to a mobile device that decodes the content. The video decoder is often implemented as a main loop which is executed repeatedly, reading encoded frames, decoding them, and writing them to an output device (e.g., a screen). The application has to deliver a prespecified throughput (e.g., 25 or 30 frames per second), which imposes a time constraint on each loop iteration. Otherwise, the movie will stutter

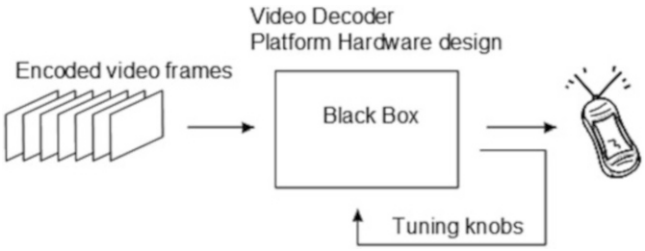


Fig. 2.2 Motivating example, no use of scenarios

and the user's experience, i.e., the QoS, will degrade. When this kind of video decoder is implemented in a mobile device that is battery-operated and thus energy-constrained, the goal of using system scenarios during the design is to reduce the energy consumption. That has to happen while retaining an acceptable QoS, including the frame rate. While every frame must be decoded within a fixed period of time, the actual time and energy needed to decode a frame on a processor with a given speed will vary significantly due to the dynamism exhibited by the application. Some frames require all the available decoding time while others demand only a fraction and leave the processor idle for the remaining time. On a small set of video sequences, we already noticed differences up to 450% in the required energy for decoding a single frame.

One well-known approach for reducing energy consumption in this situation (fixed deadline, varying decode time) is the dynamic voltage and frequency scaling (DVFS) technique [39]. When scaling the voltage, the processor's frequency and therefore the execution time scales linearly ($f_{CLK} \tilde{-} V_{DD}$), while the energy consumption scales approximately quadratically ($E \tilde{-} V_{DD}^2$). As such, DVFS gives the system an effective configuration knob, namely a choice to work at a certain frequency/voltage. That knob needs to be tuned at run-time. An important issue when applying DVFS in this situation is the need of knowing how many cycles (the cycle budget) are needed for decoding a frame, before actually decoding it. This is necessary to choose the appropriate scaling factor, i.e., to choose in which position to turn the DVFS system knob.

Existing DVFS systems work either fully dynamically based on run-time information or fully statically based on design-time analysis. More information and related work on this will be provided in Chap. 5. In the fully dynamic approach, no information about the decoder is considered except the notion of consecutive frames with different decode times that need to be decoded within a given deadline (Fig. 2.2). Without any information on how the internals of the decoder cause this variation in decode time, it is only possible to predict at run-time the required cycle budget of the current frame based upon the cycle budget needed for previously decoded frames [8, 36]. Another fully static approach considers complete information about the program at design-time and uses static analysis to determine the remaining worst-case cycle budget needed to complete execution at several

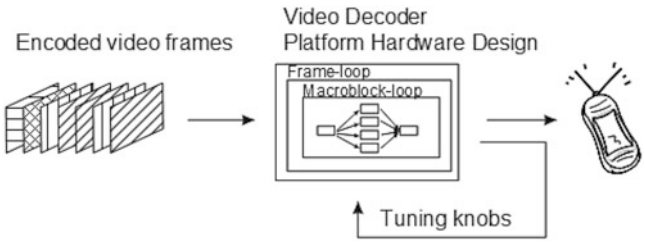


Fig. 2.3 Motivating example, system scenarios

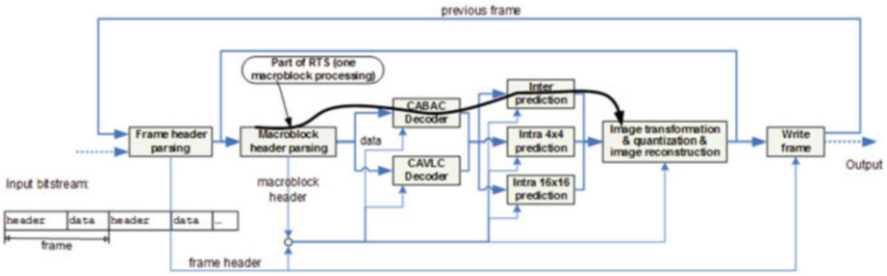


Fig. 2.4 Gray-box model of the H.264/AVC decoder processing a stream object

points in the execution and fixes the DVFS system to the corresponding voltage and frequency [62].

When using our proposed system scenarios (Fig. 2.3), we consider both information about the system at design-time and the occurrence of certain types of input at run-time, which result in particular (groupings of) RTSs. Looking at the general structure of the H.264 decoder (Fig. 2.4), we see that each frame is subdivided into blocks of 16 by 16 pixels, called macro-blocks. The main loop, which is the frame decoding loop, contains a second loop that iterates over the consecutive macro-blocks. The read part of this loop takes an encoded macro-block object from the input stream and separates it into a header and the object's data. The write part places the decoded macro-block in the frame. The decoding part consists of several kernels. Each macro-block can be encoded using a different encoding scheme determining which kernels are used. Depending on the exact breakup of how many macro-blocks of each frame belong to each scheme for a given iteration of the main frame loop, each of these kernels is executed for a certain number of times. This forms the run-time situation (RTS) which can be characterized by the current RTS parameters, i.e., the macro-block breakup in this example. These parameters can be used to predict in advance the multi-objective cost dimensions associated with the current RTS.

However, considering each possible breakup of a frame for tuning the system at run-time would cause a too large run-time storage and execution overhead. When decoding CIF images (352 by 288 pixels), consisting of 396 macro-blocks that may belong to 21 different encoding schemes, up to 6.22×10^{33} possible RTSs

would need to be considered, and for each RTS the receiver has to store the optimal frequency/voltage. This is clearly impossible for any practical system realization. Therefore, it is instead necessary to cluster frames with a similar breakup of macro-block types over the encoding schemes and mapping on the target platform. So frames that need similar cycle budgets can be safely merged into the same system scenario. For each system scenario, we then determine the optimal knob setting of the mapping scheme and the platform. For example, we can determine the frequency/voltage setting of the DVFS scheme by using the cycle budget needed for a single (worst-case) representative frame as the budget needed for all possible frames belonging to this scenario. The above has to be generalized for the multi-objective cost spaces which we typically have to deal with and that is feasible by using Pareto curve trade-off concepts [51].

At run-time, whenever a client receives a video stream from a content provider, it predicts for each frame the scenario it belongs to. Then, it scales the voltage and frequency according to the values determined and stored at design-time, thereby reducing the energy consumption while still safely meeting the deadlines. Clearly, the more scenarios are considered, the higher the energy reduction that can be obtained, but also the higher the complexity and overhead of the prediction step. The prediction causes run-time overhead and it will, e.g., add to the energy usage. The cost and gain of extra scenarios have to be traded off carefully to arrive at an optimal system realization.

Another interesting issue is to what extent scenario prediction can be made or needs to be made conservative in order to meet hard parametric constraints, e.g., on timing. For complexity reasons, it will clearly not be possible to consider all possible RTSs in the scenario definition. For the H.264 decoder, for example, there are too many RTSs to take them all into account explicitly. When at run-time a frame arrives with a previously unseen macro-block breakup, it needs to be decided what to do. Hard guarantees on system performance can be given by predicting for these new breakups the backup scenario, which is the scenario that requires the overall worst-case number of cycles to execute. Obviously, if we have to fall back frequently to this backup scenario, this will normally lead to (much) less energy reduction than potentially possible. Fortunately, by definition, this fallback will have to be used very rarely in our approach, making the average loss over the entire execution of the application minimal.

In addition, we can go one step further. In particular, since for video decoding a small percentage of missed frame deadlines is usually acceptable, one can aim for a more aggressive prediction, introducing the risk that a frame may get miss-predicted as belonging to a scenario which has a lower cycle budget than the frame really needs, causing a missed deadline. This leads to a trade-off between overall system quality and QoS in terms of missed deadlines and energy savings. For example, in [31], the application of this additional flexibility has allowed to reduce energy consumption of the H.264 decoder by 46% with less than 0.1% of the deadlines missed, by using only 32 scenarios.

To exemplify the already introduced difference between use-case and system scenarios (see Sect. 2, let us consider a mobile device running an H.264 decoder

that supports two different frame resolutions). From the user perspective, each resolution may be considered as a use-case scenario, because the resolution affects the perceived quality. Due to the different resolution, the two use-case scenarios contain a different number of macro-blocks. Each of the two use-case scenarios can be divided automatically into more system scenarios based on the frame's macro-block mapping breakup, as presented above. This breakup is of no direct interest to the application designer or final product user, because it is a system-internal artifact of the video encoding, but it can clearly be exploited to further reduce energy consumption in the mapping. It may even be possible to integrate certain macro-block breakups of the two different resolutions into a single system scenario.

4 Basic Concepts and Terminology

The goal of the system scenario method is, given a system, to exploit at design-time its possible RTSs, without getting into an explosion of detail. If the environment, the inputs, and the hardware architecture status would always be the same, then it would be possible to optimally tune the system to that particular situation. However, since most parameters are typically changing all the time, the system behaves in a dynamic way. This implies that to meet the (at least partly) strict system-level requirements which typically apply in our target application domains, it must be designed to handle also the worst-case situation for the hard QoS restriction. Still, it is possible to tune the system at run-time (e.g., change the processor frequency/supply voltage), based on the actual RTS. Unfortunately, if this has to be decided entirely at run-time, the overhead is most likely (way) too large. So, a much better approach is that an optimal configuration for each of the possible situations in the given system is selected up front, at design-time. However, if a different configuration would be stored for every possible RTS, a too huge database is required. Therefore, in our approach the RTSs similar from the resource usage and mapping perspective are clustered together into a single scenario, for which we store a single tuned configuration for the worst case of all RTSs included in that scenario. The latter is the main motivation to keep the RTSs within a given system scenario sufficiently close in the multi-dimensional cost space. For that purpose, a multi-dimensional distance metric has to be defined (see below).

The system scenario methodology deals with two main challenges. First, scenarios introduce various overheads due to switching between scenarios, storing code for a set of scenarios instead of a single application instance, predicting the RTS, etc. The decision of what constitutes a scenario has to take into account all these overheads, which leads to a complicated problem. Therefore, we divide the scenario approach into steps. Second, using a scenario method, the system requires extra functionality: deciding which scenario to switch to (or not to switch), using the scenario to change the system configuration, and updating the scenario set with new information gathered at run-time.

Many system parameters exist that can be tuned at run-time while the system operates, in order to optimize the application behavior on the platform which it is mapped on. We call these parameters system configuration knobs, in short system knobs. A large variety of system knobs is typically available in practical system applications and platforms. Section 3 has given the example of the DVFS knob; entirely different examples of other possible system knobs include the version of the code to run in case of an application that contains multiple versions of its source code; different compiler optimizations being applied to each of them [50]; and the configuration of processing elements (e.g., number and type of function units) in a multi-processor system [60]. Anything that can be changed about the system during operation and that sufficiently affects system cost (directly or indirectly) can be considered a system knob. Note that these changes do not have to occur at the hardware level; they can occur at the software or middleware levels as well. A particular position or tuning of a system knob is called a knob position. If the knob positions are fully fixed at design-time, then the system will always have the same fixed, worst-case cost. By configuring knobs while the system is operating, the system cost can be affected. In the DVFS example, the knob position is the choice of a particular operating voltage, and its change directly affects the processor speed and instantaneous power, and indirectly the total energy consumed to execute the application. However, tuning knob positions at run-time also introduces overhead, which should be taken into account when the net system cost gain is computed.

Instead of choosing a single knob position at design-time, it is possible to design for several different knob positions, combined in a set. At different occurrences during run-time, one of these knob positions is chosen from the available set, depending on the actual RTS. When the RTS starts, the appropriate knob position should be set. Moreover, the knob position is typically not changed during the RTS execution to reduce the overhead. This means the RTS is the integral primitive and hence it also implies the unit time division in our approach. Therefore, it is necessary to determine which RTS is about to start. This prediction is based on RTS parameters, which have to be observable and which are assumed to remain sufficiently constant during the RTS execution. These parameters together with their values in a given RTS form the RTS snapshot. In the H.264 example, the RTS corresponds to the decoding of a frame, and the RTS parameter is the frame breakup into the macro-block types. The number of distinguishable RTSs from a system is exponential in the number of observable parameters. Therefore, to avoid the complexity of handling all of them at run-time, several RTSs are clustered into a single system scenario. A trade-off is present here between optimization quality and run-time overhead of the scenario exploitation, as already illustrated in the motivating example of Sect. 3.

At run-time, the RTS parameters are used to detect the current system scenario rather than the current RTS. In principle, the same knob position is used for all the RTSs in a scenario, so they all have the same cost value, as already indicated: the worst case of all the RTSs in the scenario. Therefore, it is best to cluster RTSs which have nearby cost values in the multi-objective trade-off space. Since at run-time any RTS may be encountered, it is necessary to design not one scenario but rather a

scenario set. A scenario set is a partitioning of all possible RTSs, i.e., each RTS must belong to exactly one scenario.

The approach presented above is relatively easy to apply when the cost space is one-dimensional, i.e., when one cost aspect dominates or when all the different cost aspects have been combined in a normalized weighted sum. The latter is not always easy and usually quite suboptimal in practice because comparing apples and oranges in a single dimension usually leads to inconsistencies and suboptimal results. Hence, N-dimensional Pareto sets can and should be used to specify the costs of a system scenario consisting of different RTSs instead of a weighted one-dimensional cost dimension. Such Pareto sets [24, 51] allow to work with a Pareto boundary between all feasible and all non-feasible points in the N-dimensional cost space. The Pareto boundary (the Pareto points) for all the possible RTSs that have been clustered into a scenario (and that can potentially be encountered at run-time) characterizes the scenario. Unfortunately, it becomes less obvious to deal with statements like nearby cost values of RTSs or taking the worst case of all the RTSs in the scenario. So, similarity between costs of different RTSs or in general sets of RTSs (scenarios) has to be substituted by a new element, e.g., by defining the normalized, potentially weighted distance between two N-dimensional Pareto sets as the size of an N-dimensional volume that is present in between these two sets. Based on this distance value, closeness of potential scenario options can be characterized, e.g., to decide whether or not to distinguish scenarios. For more details, the reader is referred to [48, 90, 91].

Another crucial characteristic of the system scenario approach lies in the usage for applications dominated by RTS parameters governed by control variables or data variables in the application source code. Initially we have focused our methodology and techniques only on control variables [29]. These techniques assume that the parameters are control variables and/or that they have a limited number (e.g., a few tens up to maximally 100) of possible parameter values for each parameter individually. They can then make use of enumeration and apply a bottom-up approach to cluster these values into system scenarios [26, 28]. This implies some form of conditional CASE statement can be used to represent the options and to select the active case during execution. This is illustrated in Fig. 2.5 on the left. Here 2 different control variable parameters ξ_i lead to in total 9 C_{ij} run-time situations. The latter have been clustered in 3 system scenarios S_j. The control diagram represents both the way the detection and the switching can happen at run-time. The discussion in most of this chapter uses this type of systems as a basis. So we will expand in depth the theory for this case. As a main illustration in this chapter, we will also use a running case study based on control variable system scenarios, namely the H.264 decoder.

Later, after 2010, we have extended this approach further to data variables [33]. The main theory and illustrations are provided later in this book, in Chap. 3. But here already a brief introduction is included. In particular, this data variable alternative implies that systems have parameters with widely varying data-dependent values coming from data-dependent loops or recursions. When the RTS parameters are data-dependent in this way, they may have thousands or even millions of possible

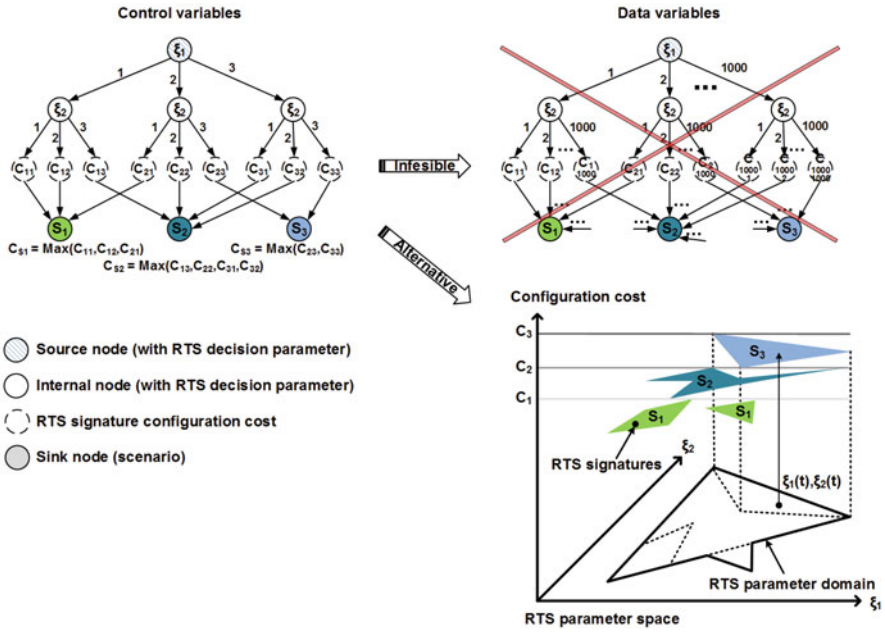


Fig. 2.5 Control vs data variable-based scenario identification

data values, making bottom-up clustering and enumeration-based prediction/detection totally impractical. That is also shown in Fig. 2.5 at the top right side. The same figure at the bottom right illustrates the theoretical concepts of how to deal with such data variables (see [33]). Let us start from k RTS parameters and N profiled RTS signatures. The latter are defined for each RTS i in Eq. (2.1).

$$r(i) = \xi_1(i), \xi_2(i), \ldots, \xi_k(i); c(i) \tag{2.1}$$

If we assign one dimension to each RTS parameter, the resulting k-dimensional space will define all theoretically possible values for the RTS parameters in the application. We will again call such space an RTS parameter space. When static max and min constraints on the values are added, the space reduces to one or several k-dimensional domain(s). With the representation above, the scenario identification task can be viewed as a distribution of points into S different groups, representing system scenarios, according to which the overall configuration cost is minimized. An RTS point i is assigned to scenario j whenever its cost $c(i)$ falls into that scenario's cost range $[C(j)_{\min}, C(j)_{\max}]$. The scenario cost ranges are determined by a balancing function that tries to achieve that all scenarios have a near-equal probability to occur at run-time. In this way, rare system scenarios are avoided since their storage cost will exceed the gains of adding them. We measure this probability by the number of points, including the repeating ones, that each scenario

contains and call it scenario size. Given a list r of sorted cost RTS signatures (e.g., descending), the scenario cost ranges are then given by the indices corresponding to the integral number of maximum and minimum scenario sizes N/S, as represented in Eq. (2.2).

$$C(j)_{\max} = r((j-1) * N/S + 1); C(j)_{\min} = r((j) * N/S) \tag{2.2}$$

The cost of scenario C_j is defined as the maximum cost of RTS signatures that it includes: $C_j = C(j)_{\max}$. As illustrated in Fig. 2.5, the projection of scenarios onto the RTS parameter space will produce $M \leq S$ regions that characterize the system scenarios in terms of RTS parameter values. Each region can be described as a polyhedron, and the run-time scenario prediction can be done by checking which polyhedron contains the RTS parameter values of the next RTS. Since we know which scenario the region belongs to, we can foresee that the next running cost will be no more than the cost of that scenario. Checking if a point lies inside a polyhedra is the classical data point location problem from the computational geometry domain, and the advantage of using it for prediction is that it operates on/stores only the vertices of polyhedral regions, not the whole RTS parameter space. This is in contrast to the multi-valued decision diagram (MDD) concept which is used for the control variable option (see Sect. 5.3).

This top-down data variable oriented approach can hence handle arbitrary large domains, provided that the number of convex polyhedral regions stays reasonably low. Otherwise prediction overhead will grow unacceptably. The number of regions depends on the number of system scenarios and the underlying structure of the system, i.e., the relationship between the cost locality of RTS points and the value locality of their RTS parameters.

The desired number of system scenarios is best defined by the user according to the characteristics of the application domain. Typically this is limited to a few dozens because beyond that the potential gains in better following the system dynamics are counterbalanced with the additional cost complexity of detecting and exploiting the (too) large set of possible system scenarios. This can be easily motivated by considering that if the scenario ranges are more or less equally spaced (which is one of the aims anyway) in a one-dimensional cost space, a few dozen scenarios already lead to less than 5% distinction in the associated cost function. For cases where the number of dimensions grows to a large amount, it becomes more complex and the required number of system scenario sets to cover this sufficiently well could potentially grow to a huge amount. But when this would happen, then the extension of so-called subscenarios which we have first introduced in [72] will still provide a way out of this undesired explosion.

Another important extension to keep the number of system scenarios limited is to explicitly allow some remaining parametrization in the position of the Pareto curve in the cost function space. As indicated above, when the system knobs are changing at run-time, we anyway do not want to store a different scenario execution for each of these system knob instances. So we will allow a given "mapping" from the functional system level up to the system implementation level (where the cost functions are defined) to become parametrized in terms of system knob or RTS

parameters. However, that should only be allowed for parameters which maintain the good quality of the chosen mapping for all of the RTS parameter and system knob settings covered by the scenario. This quality will be defined in terms of a multi-dimensional distance function allowing to evaluate how "close" RTS instances are positioned, as explained earlier. We have to be careful then that only parameters are allowed which do not affect the optimal mapping choice and which maintain a simple near-linear relation in defining the related cost function. For instance, when you map a given one-dimensional loop nest of size N with the available control variable options to a data-path or in general a digital SoC platform, the value of N only needs to be fixed inside the control variable scenario when that way of platform mapping is more than negligibly depending on the value of N. But in many applications that is not the case, or at least not for all loop iterators inside a multi-dimensional loop nest. So then the resulting system scenarios can share the same internal mapping for each RTS instance and still have a cost function (defined, e.g., in terms of performance, energy, or sometimes area) which can be expressed as a simple parametrized formula directly depending on the value of N. For instance, performance and energy for the execution of the entire loop can be considered linear in N. Something similar can happen for data variable scenarios when you consider control flow or other parameters that are not related directly to the data variable dimension (which are usually loops or data sizes). This implies that the distance function calculation for the Pareto curves in the cost space defined above should also incorporating N as a parameter. Hence, the closeness of RTS instances that are candidates for the system scenario clustering is evaluated for a given value of N. To allow the easy extrapolation for the closeness for all other values of N, it is now obvious why we strongly preferably want a linear relation of the cost function axes. A clear demonstration of the power and the importance of such parametrized formulas is present in the scenario methodology that has been conceived for dealing with the mapping of dynamic wavelet-based video codecs on L1 data caches (see [16, 17]). There, quite complex but run-time efficient equations have been derived to combine many different Pareto curve working points and positions in compact representations which are tightly characterizing the actual run-time situations. The case of even more strongly dynamic video codecs has been dealt with in [18], with a case study focused on motion compensated temporal filtering (MCTF). In that application, the parameters are so strongly varying at run-time that the combination of the Pareto-curves for the RTSs cannot happen only based on equations. So an alternative has been proposed for such cases based on specializing for the most promising localization schemes.

The following section details the systematic aspects of our methodology of identifying and exploiting system scenarios to create more efficient embedded systems, describing generic solution strategies for the various methodology steps whenever possible.

5 System Scenario Methodology

Although the concept of system scenarios has been applied before on top of concrete design techniques both in an ad-hoc way [9, 34, 47, 60] and in a more systematic way [25, 28, 31, 44, 50, 85], it is possible to generalize all those scenario approaches into a common systematic methodology. The first paper where we have described this is [29]. This section summarizes that general and still near-optimal methodology, providing generic solutions whenever possible. But the main instantiation is worked out for control variables, similar to what has been presented earlier in [29]. On top of this, we have further extended the approach to cover data variables, as already introduced in Sect. 4 and later in Chap. 3. The rest of this section is structured as follows. The overall methodology overview is provided in Sect. 5.1. The remaining subsections refine each of the steps of the general methodology. In the subsequent subsections, we always refer to system scenarios also when we use the abbreviated term scenario.

5.1 Methodology Overview

Even though the system scenario concept is applicable to many contexts, we have devised a general methodology that can be instantiated in all of these contexts. This system scenario methodology and the presented generic solutions for some of its steps deal with issues that are common: choosing a good scenario set, deciding which scenario to switch to (or not to switch), using the scenario to change the system knobs, and updating the scenario set based on new information gathered at run-time. This conceptually leads to a five-step methodology (Fig. 2.6), each of the steps having a design-time and a run-time phase. The first step, identification, is somewhat special in this respect, in the sense that its run-time phase is merged into the run-time phase of the final step, calibration.

1. Identification of the scenario set: In this step, the relevant RTS parameters are selected and the RTSs are clustered into scenarios. This clustering is based on

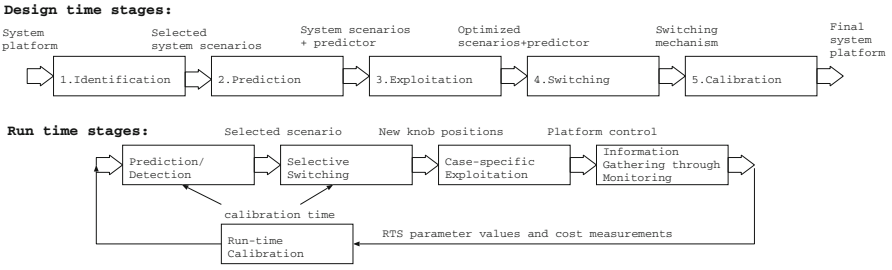


Fig. 2.6 Overall flow of system scenario methodology

the cost trade-offs of the RTSs, or an estimate thereof. The identification step should take as much as possible into account the overhead costs introduced in the system by the following steps of the methodology. As this is not easy to achieve, an alternative solution is to refine the scenario identification (i.e., to further cluster RTSs) during these steps. Section 5.2 discusses the identification step in more detail.

2. Prediction of the scenario: At run-time, a scenario has to be selected from the scenario set based on the actual parameter values. This selection process is referred to as scenario prediction. In the general case, the parameter values will not be known before the RTS starts, so they will have to be estimated. Prediction is not a trivial task: both the number of parameters and the number of potential scenarios may be relatively large, so a simple lookup in a list of scenarios may not be feasible. Still, as already mentioned we do aim at a few dozen scenarios in practice so this step will also not explode. The prediction incurs a certain run-time overhead, which depends on the chosen scenario set. Therefore, the scenario set may be refined based on the prediction overhead. Section 5.3 details the two decisions made by this step at design-time: selection of the run-time prediction algorithm and refinement of the scenario set.
3. Exploitation of the scenario set: At design-time, the exploitation step is essentially based on some optimization that is applied when no scenario approach is applied. A scenario approach can simply be put on top of this optimization by applying the optimization to each scenario of the scenario set separately. Using the additional scenario information enables better optimization. At run-time, the exploitation is in fact the execution of the scenario. Section 5.4 details the common problems that should be handled during the exploitation step.
4. Switching from one scenario to another: Switching is the act of changing the system from one set of knob positions to another. This implies some overhead (e.g., time and energy), which may be large (e.g., when migrating a task from one processor to another). Therefore, even when a certain scenario (different from the current one) is predicted, it is not always a good idea to switch to it, because the overhead may be larger than the gain. The switching step, detailed in Sect. 5.5, selects at design-time an algorithm that is used at run-time to decide whether to switch or not. It also introduces into the application the way how to change the knob positions, i.e., how to implement the switch, and refines the scenario set by taking into account switching overhead.
5. Calibration: The previous steps of our methodology make different choices (e.g., scenario set, prediction algorithm) at design-time that depend very much on the values that the RTS parameters typically have at run-time: it makes no sense to support a certain scenario if in practice it (almost) never occurs. To determine the typical values for the parameters, profiling augmented with static analysis can be used. However, our ability to predict the actual run-time environment, including the input data, is obviously limited. Therefore, we also foresee support for infrequent calibration at run-time, which complements all the methodology steps previously described. At design-time, information gathering mechanisms are designed and added to the application. At run-time they collect information

about actual values of the parameters and the quality of the resulting system (e.g., number of deadline misses). Besides this, a calibration mechanism is introduced in the application. This is used to calibrate the cost estimates, the set of scenarios, the values of the parameters used for scenario prediction, and the knob positions. Calibration of the scenario set should not take place continuously during run-time, but only sporadically, at appropriate well-selected calibration moments in the application execution. Otherwise the overhead would obviously become too large. Also note that calibration is not meaningful when all quality constraints are hard. It can only be applied if at least some constraints are soft, or to optimize average-case behavior in the absence of hard constraints (e.g., when optimizing memory usage for energy reduction). Section 5.6 presents techniques for calibration.

In the following paragraphs, we indicate intuitively why, in the design-time part of the methodology, the steps have been ordered as proposed. In particular, the reasoning behind this is based on a gradual pruning of the possible final scenario decisions. First, during identification, RTS parameters are limited to the ones that have a sufficient and observable cost impact on the final system. Then during clustering, we select the parameters that are easiest to be controlled as the actual system knobs and we cluster the corresponding RTSs based on similarity of cost (and knob settings, if applicable). In this way, we ensure that the cost distance between any two scenarios is maximized. This is needed because we have a clear trade-off between the gains by introducing more scenarios and the cost it involves.

This trade-off leads to a further pruning of the search space for the most effective final scenario decisions. In the prediction step, we have to limit the potentially most usable scenarios to the ones that are also predictable at run-time with an affordable overhead. Also here a global trade-off between gain and cost (run-time prediction overhead) is present. We cannot perform this second step of our method prior to the identification step because we cannot sufficiently accurately estimate the prediction cost before we at least have some (good) idea about the clustering of RTSs in scenarios. Note that the opposite is not true: the information of the prediction step is not essential to decide on the clustering. This creates an asymmetrical relation which is the basis for the split between the two steps (see also the constrained orthogonalization approach in [7]).

Only when we have decided how to perform the prediction, we can start the exploitation of the resulting scenarios in the particular application domain (step 3). Indeed, we could already start the exploitation after having the first clustering step, but that is not always efficient: the knowledge of the prediction cost will give us more potential for making suitable exploitation decisions. In contrast, the knowledge of the exploitation itself is not yet needed to make a good pruning choice on the prediction related selection.

Finally, in the proposed design trajectory, we only decide on the scenario switching based on the actual overhead that is involved in the switching. The latter is only known after we have decided how to exploit the scenarios. The calibration step can be applied only when the rest of the steps are already decided, as information

about the scenario set, and the prediction and switching algorithms are needed to design the information gathering and calibration mechanism.

So every step of our design-time methodology is positioned at a location where it has maximal impact but also where the required information to effectively decide on it is already available as much as possible. The proposed split up in steps and order prevents detrimental phase coupling to a very large extent. This avoids iteration on any of the individual steps after completion of a subsequent step in the methodology, which is a deliberate and important property of our generic design methodology. The ordering of the steps at run-time follows the natural ordering of the various activities as they are needed at run-time. The ordering is in line with the design-time ordering with two exceptions. The first one was already mentioned: the identification and calibration steps are integrated, because part of the run-time calibration step may be to identify new scenarios, and no other means to identify new scenarios at run-time are available. Secondly, the order of switching and exploitation is reversed when compared to the design-time order, as the run-time switching prepares the system for the given exploitation.

The next subsections detail the various steps of the methodology, outlining generic solutions whenever such solutions are possible. All the presented solutions for control variable-based system scenario exploitation have been largely automated, and are as much as possible applied in the various case studies presented in the rest of this book. The wide range of applicable demonstrators is also briefly summarized in Sect. 6.

5.2 Identification

Before gaining the advantages that a scenario approach gives, it is necessary to identify the different scenarios that group all possible RTSs. This identification process happens in two phases. First the interesting snapshot parameters are discovered. As mentioned before, a snapshot contains all parameters as well as their values that characterize a certain RTS. However, we are only interested in those parameters which have an impact on the system's behavior and execution cost. For example, interesting RTS parameters for an audio-video system are the size of the video frame, and whether the audio stream is mono or stereo. The values of the selected parameters will be used to distinguish between the different RTSs, so two RTSs with the same snapshot are considered identical. However, they may still have different actual cost values in the multi-dimensional objective space, due to a choice of the parameters that does not precisely capture all the unique system behaviors. For example, two RTSs with a different data-dependent loop bound have a different execution time, but we consider them the same RTS if we are not observing that loop bound. When we are also observing that loop bound, each number of iterations corresponds to a different RTS. In the general case, a parameter selection that does not precisely capture all the individual behaviors of a system may lead to RTSs with a set of Pareto points in the multi-dimensional cost space as their actual (worst

case) cost values. Such a parameter selection may be present due to an imperfect analysis or unavoidable for complexity reasons. However, it may also be deliberate, e.g., with the intention to handle certain minor dynamic variations or configuration options (i.e., a choice among different knob positions) entirely at run-time, or to create a hierarchy of scenarios, where the variation within one scenario at a certain hierarchical level is handled by another set of (sub-)scenarios.

Following the parameter discovery, all possible RTSs are clustered into system scenarios based upon a multi-objective cost function. The multi-objective cost function is dependent on the specific optimization and the system knobs we have in mind for the exploitation step. For the H.264 decoder presented in Sect. 3, we aim at reducing energy by applying DVFS and so we need accurate cycle-budget estimations for decoding the frames. The objective function in this case is one-dimensional and it is represented by the cycle budget needed for decoding each frame. Note that the decoding of a frame was considered the RTS in this example. The remaining part of this section details the two phases of the identification process.

5.2.1 RTS Parameter Discovery

In related work done so far, usually, parameter discovery is performed in an ad-hoc manual manner by the system designer, by analyzing the application and profiting from domain knowledge. This is fine when all the important parameters are immediately obvious, such as the frame size in a video decoder. However, this process will in practice prove tedious and incomplete for complex systems, as parameters that may have a large impact on the system behavior will easily go unnoticed. Developing a fully general tool that discovers the interesting parameters for all the design approaches where scenarios may be applied is very hard to realize due to the diversity of cost functions and optimization objectives. Therefore, we have developed a set of quite broadly applicable domain-specific approach that is presented in the remaining part of this subsection. We will only illustrate this here for the control variable case. More information on the data variable alternative is provided in Chap. 3.

Our algorithm searches for control variables in the application source code that have a certain impact on the application resource requirements (e.g., number of cycles, memory utilization). These parameters fulfill the two requirements for selection: they are observable and they influence the application's behavior and cost (i.e., the resource needs). A first version of this tool [25, 27] statically analyzes the application source code to identify these variables. It is applicable for hard (real-time) constraints, due to the conservative analysis. On top of this, in [28], a version applicable for soft real-time systems has been presented. It profiles the application, and it uses the collected information for eliminating those control variables whose values do not have a real impact on system cost. This is illustrated in Fig. 2.7.

In each step of the design, and at each iteration, manual intervention is possible, but not necessary, to steer the decision process. The final result of this automated

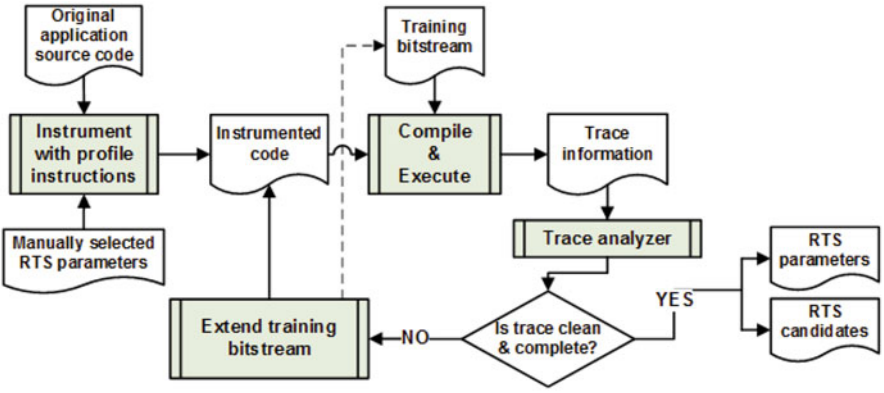


Fig. 2.7 RTS parameter identification approach

analytical discovery is thus a list of relevant RTS parameters. During the profiling step it is of course possible to collect additional information such as the met RTSs identified by their snapshot, together with their cost. This information is then used in the RTS clustering step. However, finding a representative training bit-stream that covers most of the behaviors that may potentially appear during the system lifetime, particularly including the most frequent ones, is in general a hard problem. Hence, in contrast with analysis-based identification that covers all possible RTSs, the profiling-based identification is not conservative. It can happen that, at run-time, when the system runs, an RTS that was not considered during identification is met. Therefore, a way of handling this situation should be added in the final implementation of the system. The calibration step in our methodology (see Sect. 5.6) has been included for this reason, among others. Experiments show that the combination of profiling-based parameter discovery and calibration is quite robust, alleviating the problem of finding representative training sets and reducing the time needed for training.

5.2.2 RTS Clustering

Next, using the discovered RTS parameters, all identified RTSs are clustered into a set of system scenarios. For each RTS, in the most general case, we have a multi-dimensional Pareto surface of potential exploitation points in the multi-dimensional exploration space. The clustering is done based upon a multi-objective cost function which is related to the specific trade-off exploration we want to apply to the system. Hence, the clustering searches for RTSs with similar Pareto surfaces. It starts from RTS snapshots and generates a set of scenarios, each of the scenarios being identified by a set of snapshots. The clustering takes into account the following information:

1. how often and for how long each RTS occurs at run-time,
2. the cost deviation that occurs when clustering multiple RTSs into a single scenario,
3. how many switches occur between each pair of scenarios, and
4. the run-time scenario prediction, storage, and switching overhead.

Having a multi-dimensional cost function means that both the inherent scenario costs and the switching cost become multi-dimensional also. Creating a good scenario set under these constraints can be formulated as an N-dimensional optimization problem. However, this optimization problem does not have a general practically executable solution, so heuristics need to be developed.

When clustering different RTSs into a scenario, we determine the cost of the scenario as the maximal cost of the RTSs that compose the scenario (which in a multi-dimensional cost space results in a Pareto surface). The clustering process is driven by two opposing forces. One force drives towards a large number of scenarios that contain a few RTSs, the extreme being each scenario to contain only one RTS, by only grouping RTSs with similar cost together, so that the estimated deviation between the cost value of an RTS and the cost of the scenario remains small. It uses the information from items (1) and (2) of the list above. The other force takes into account the overheads (items (3) and (4) of the above list) introduced by the existence of a large number of scenarios, and it aims at decreasing the number of scenarios by increasing their size in the number of contained RTSs.

Since the application does not remain in the same scenario forever, the switching overhead has to be taken into account also. This overhead usually has non-negligible effects on the cost function (e.g., scaling frequency and voltage of the processor costs both time and energy). So, depending on how large the switching overhead is, the aim is to reduce the number of scenario switches that appear at run-time. Taking this into account, the two forces identified above have to generate a trade-off by clustering together into a scenario, not only RTSs with similar cost, but also the ones between how many switches appear at run-time.

The storage overhead of scenarios is strongly dependent on the kind of optimizations that are applied in the exploitation step. For example, in the H.264 decoder presented in Sect. 3, a table has to be kept which maps the different scenarios to the optimal frequency-voltage pair. When the number of scenarios increases so does the size of this table, but the overhead per scenario will be small. On the other hand, in the case study presented in section x.y, when optimized code is generated for each separate scenario, the overhead for storing this scenario-specific code is rather large.

Finally, since the scenarios need to be predicted at run-time, also the scenario predictor should be considered. If the amount of scenarios increases, it will result in a larger and perhaps slower predictor. Also, the probability of a faulty prediction may increase with the number of possible scenarios.

To incorporate the potential effect of all these additional side effects and to avoid phase coupling and expensive iterations in the flow, we have developed appropriate estimates which can provide safe bounds upfront of the other steps, during the identification/clustering step [86, 87].

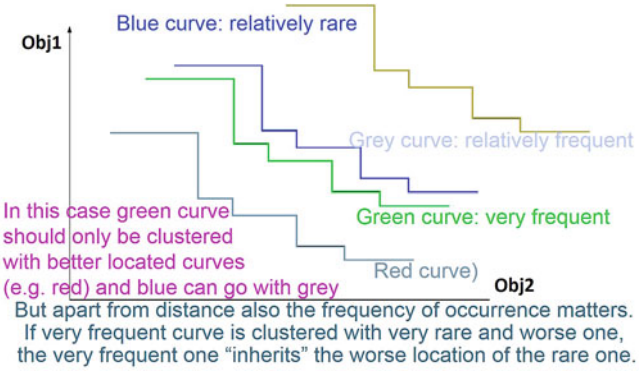


Fig. 2.8 Illustration of RTS clustering

Figure 2.8 depicts a clustering example with a two-dimensional cost function for a generic multi-dimensional control or data variable context. Each RTS is represented by a (two-dimensional) Pareto boundary which represents different RTS knob configurations. Usually, the RTSs with similar Pareto boundaries are clustered to one scenario at design-time. Thus, the distance between two Pareto boundaries determines which ones to cluster. The cluster is then represented by its worst-case Pareto boundary. As already mentioned, apart from the distance also the frequency of occurrence is important. If a very frequent RTS is clustered with a very rare one, which has a worse Pareto boundary, this scenario and all RTSs in this scenario inherit the worse, rare Pareto boundary. In such a case, it is better to cluster the frequent Pareto boundaries with better rare Pareto boundaries so that the frequent Pareto boundary represents the created scenario. Based on these two criteria, an illustration is provided in Fig. 2.8. There, boundaries "Green curve" and "Red curve" are clustered in forming the first scenario, and boundaries "Grey curve" and "Blue curve" from the second scenario cluster. This is so even though the distance between "Green curve" and "Blue curve" is quite small; so it would seem logical (without the frequency of occurrence info) to cluster them together. This analysis and clustering are done at design-time. At run-time, the appropriate scenario with its Pareto boundary is identified and a concrete Pareto point is selected which is related to a specific knob configuration.

5.3 Prediction

This step aims at deriving a predictor, which can determine at run-time the appropriate scenario in which the system executes. It starts from the information collected in the identification step. Just as this parameter identification step, scenario prediction can be solved in a rather generic way that is widely applicable, but only when we clearly distinguish between the control and data variable case. The material in this

generic flow chapter is reusable across both cases. But when we illustrate a concept, we will now cover only the control variable option. The data variable alternative is illustrated later in the book and basic information can also be found in [88].

The resulting predictor mainly bases its decision on the values of the RTS parameters. Moreover, it has to be flexible (e.g., to have a structure that can be easily modified during the calibration phase) and to add a small decision overhead in the final system. We can define it as a prediction function in Eq. (2.3).

$$f : \Omega_1 x \Omega_2 x \ldots x \Omega_n \rightarrow \{1; ..; m\} \tag{2.3}$$

where n is the number of RTS parameters, Ω_k is the set of all possible values of the parameter ϵ_k (including the undefined value), and m is the number of scenarios in which the system was divided. The function f maps each RTS i, based on the parameter values $\epsilon_k(i)$ associated with it, to the scenario to which the RTS belongs. If at run-time an RTS occurs that was not considered during the identification phase (e.g., because it was not met during profiling), it is mapped to the scenario that can deal even with the worst-case situation, the so-called backup scenario (as defined in Sect. 4).

A predictor based only on the prediction function approach can be applied only after all the parameter values are known. If the identification was done in a conservative mode, which covers all possible RTSs that may appear at run-time, the prediction accuracy will be 100%, and we can speak about scenario detection. Waiting until all the parameter values are known at run-time will postpone the prediction moment unnecessarily long, and the scenario will predicted too late to still profit maximally from the applied optimization. To handle this problem, multiple approaches may be considered (not necessarily in isolation), like either reducing or changing the set of considered parameters, or instead combining the prediction function with pure probabilistic prediction. In the first approach, we search for the set of parameters that can be used to identify the set of predictable scenarios that gives the highest gain, taking into the account the moment when they can be predicted at run-time. In the second case, the scenario prediction point may be moved to an earlier point in time by augmenting the prediction function with a mechanism that selects from the possible set of scenarios predicted by the function, the one with the highest probability. For example, the mechanism may use advanced branch predictors [12] or phase predictors [75].

Using the probabilistic approach, the miss-prediction rate may increase. That can be due to two problems: either over-prediction, when a scenario with a larger (multi-dimensional) cost is selected, or under-prediction, when a scenario with smaller (multi-dimensional) cost is selected. The first type does not produce critical effects, just leading to a less cost effective system; the second type often reduces the system quality, e.g., by increasing the number of deadline misses for the H.264 decoder.

The place where the prediction function is introduced into the application is called a scenario prediction point. From a structural point of view, considering the number of times and the places where the prediction function is introduced into the application, the predictors can be classified as follows:

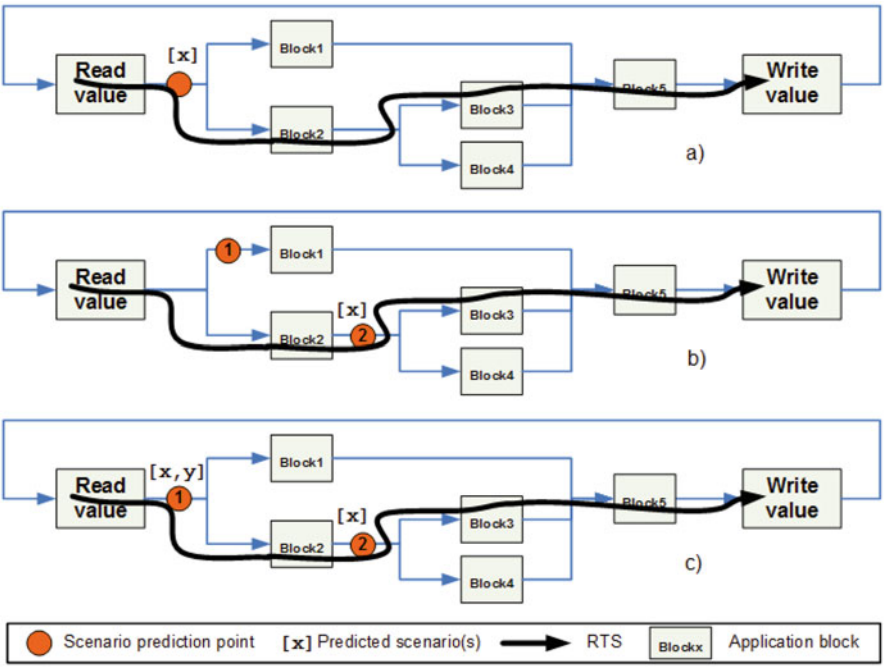


Fig. 2.9 Types of scenario prediction: (**a**) centralized predictor, (**b**) distributed predictor with exclusive points, and (**c**) distributed predictor with refinement points

- Centralized: There is only one central point in the application where the current scenario is predicted. It is inserted in the application code in a common place that appears in all scenarios. For example, in the case of the application model presented in Fig. 2.9a, it is introduced in the main loop, after the read part, when all the information necessary to predict the current scenario is known.
- Distributed: There are multiple scenario prediction points, which may be:
  - Exclusive points: An identical (or tuned) prediction function is introduced multiple times into the application, in all the places where the RTS parameter values are known. At run-time, only one point from the set is executed in each RTS. This kind of predictor solves the problem that no common place may exist in all scenarios, in order to insert a centralized predictor. Figure 2.9b depicts a case where one of two prediction points is being executed for different RTSs.
  - Refinement points: Multiple points, which work as a hierarchy, are used to predict the current scenario in a loop iteration; the first that is met at run-time predicts a set of possible scenarios, and the following refine the set until only one scenario remains. This extension can improve the efficiency of optimizations as earlier switching between scenarios may be done, but it increases the number of switches. Hence, a trade-off should be considered

when using it, which depends on the problem at hand. This is actually a very similar problem trade-off as the one for the RTS clustering substep. Hence, we believe that also here a similar set of heuristic algorithms can be reused. Usually, when switching between scenarios after a refinement predictor, the new scenario may be the scenario with the worst-case cost from the remaining set of scenarios. However, the probabilistic approach presented above could also be used to select the scenario to which to switch.

Let us look into more detail at the example depicted in Fig. 2.9c. Considering the scenario that executes kernels two, three, and five, in the first scenario prediction point, the set containing scenarios x and y is selected. Then, in the second scenario prediction point, the set is refined to only one scenario, x.

To illustrate this, let us first look specifically at the control variable case. Then, a generic implementation of a prediction function f, which incorporates requirements like flexibility and small overhead, is a multi-valued decision diagram [80], which consists of a directed acyclic graph $G = (V, E)$ and a labeling of the nodes and edges. The sink nodes get labels from $1, .., m$ and the inner (non-sink) nodes get labels from $\xi_1, \ldots, \xi_n$. Each inner node labeled with ξ_k has a number of outgoing edges equal to the number of the different values $\xi_k(i)$ that appear for RTS parameter ξ_k during the identification phase plus an edge labeled with other that leads directly to the backup scenario. Only one inner node without incoming edges exists in V, which is the source node of the diagram, and from which the diagram evaluation always starts. On each path from the source node to a sink node each RTS parameter ξ_k occurs at most once.

A simplified example of a decision diagram for the motion compensation kernel from the H.264 video decoder is shown in Fig. 2.10. When a prediction function is used, it introduces two overheads: first code size and second average run-time

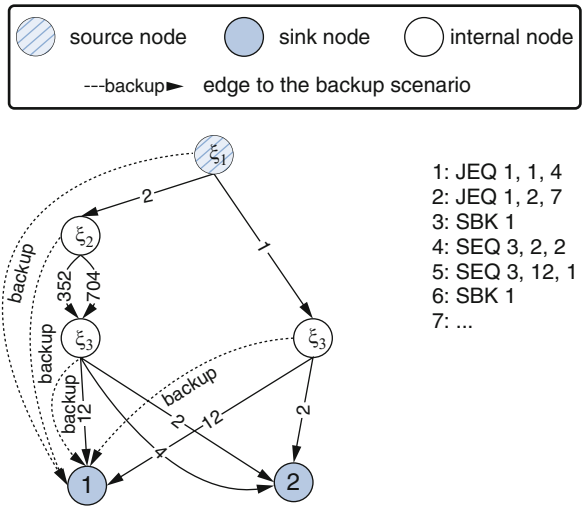


Fig. 2.10 Illustration of a decision diagram and its implementation

evaluation cost. Also this is similar to the RTS clustering substep. In the proposed solution, both depend on the decision diagram size (number of nodes and edges). Hence, reducing its size, the overheads decrease.

However, the applied reduction rules may affect the prediction quality, especially for the RTSs that were not considered during the identification step (because for these RTSs, a different scenario may be predicted instead of the backup scenario). A few reduction rules are analyzed in [28]. Note that these reductions may render certain RTS parameters redundant, which happens if they do no longer occur in the final predictor. Also note that, conceptually, decision diagram transformations are scenario refinements, because a transformation affects the parameter value ranges characterizing a scenario. We propose to implement decision diagrams as a program in a restricted programming language. More details are provided in [28].

For the data variable case, we need a partly different approach where we have to incorporate the ranges for the widely varying data-dependent parameters, as addressed in [87–89] and as further discussed in Chap. 3.

In conclusion, the above subsection has detailed a generic form and implementation of scenario predictors, that is flexible and well suited for calibration, as explained in more detail in Sect. 5.6. We end this section by mentioning several other aspects that may be addressed during the prediction step at design-time. The prediction step may cover the following actions:

1. a further clustering of scenarios considering the prediction overhead and the moment when the scenario may be predicted
2. possibly, a further pruning of the RTS parameters
3. clustering of previously unassigned RTSs (i.e., the ones that were not met during the identification process) into scenarios
4. defining and placing the prediction mechanism into the application, by trading off prediction accuracy versus overhead, which influence the final system cost and quality.

5.4 Exploitation

The exploitation step in the scenario methodology is very dependent on the application and system context in which scenarios are applied. Nevertheless, some general aspects have to be kept in mind. Exploitation in the context of the scenario methodology should be refined in two ways, to a large extent independent of the type of exploitation. First, optimizing each scenario in isolation will be inefficient. In practice, a strong correlation is present between the analysis and the optimization choices of the different scenarios, so the optimization of a scenario can be performed more efficiently by reusing information of other scenarios. Second, separate optimization for each scenario leads to separate systems. Simply putting all these next to each other will easily imply a huge overhead. Therefore, whatever

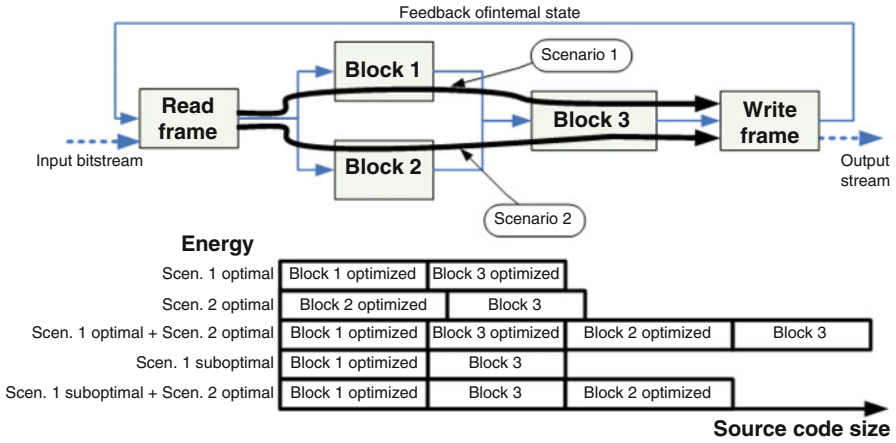


Fig. 2.11 Scenario source code merging

is common between different scenarios should be merged together, e.g., by using code compaction techniques [11, 13].

The remaining differences cause exploitation overhead, which should be taken into account to further refine the scenario set. Some optimizations that are suboptimal for an individual scenario may be optimal from the system cost perspective when considering exploitation overhead. How difficult it is to simultaneously optimize scenarios depends on the context. As an illustrative but easily generalizable example, Fig. 2.11 depicts an application with two scenarios: scenario 1 for the case where kernels 1 and 3 are executed, and scenario 2 for the case where kernels 2 and 3 are executed. To optimize the application for energy, a compiler may optimize each scenario separately to reduce the number of computation cycles. Assume that the energy-optimal exploitation of each scenario is, for scenario 1, to optimize both kernels 1 and 3, and, for scenario 2, to optimize only kernel 2, kernel 3 already being energy optimal for this scenario. Combining these two optimal scenario exploitations, the application source code contains code for kernel 3 twice (once optimized for scenario 1, and once untouched, as used in scenario 2). If the energy overhead introduced by storing the two versions of kernel 3 is large, the energy-optimal system can be obtained by using a suboptimal version of scenario 1, as presented also in Fig. 2.11. This version uses the original implementation of kernel 3, so no code duplication for this kernel is needed in the final implementation of the application.

Both mentioned exploitation refinements for scenarios are specific to the type of optimization that is performed, so exploitation cannot really be fully generalized. More examples are provided in the case study summary of Sect. 6 and in the subsequent application chapters of this book.

5.5 Switching

A system execution is a sequence of RTSs, and therefore a sequence of scenarios. At the border between two scenarios during execution, switching occurs. For executing this switch at run-time, at design-time a mechanism is derived and introduced into the system. The switching decision and process (changing the knob positions) may incur overhead, which is taken into account to further refine the scenario set.

Moreover, at run-time it should also be taken into account to decide whether or not to switch to a different scenario, together with other information (i.e., the sequence of previous and possible following RTSs). The expected gain times the expected time window where the scenario is applicable has to be compared to the exploitation cost, as already mentioned. The structure of this switching mechanism should be flexible enough to allow it to be calibrated. Even if the switching overhead is exploitation dependent, our methodology treats this overhead in a general way. It uses the scenario cost versus overhead reports (e.g., energy, time) together with the information about how often a switch between two given scenarios appears at run-time, to avoid spending most of the system's running time on switching between scenarios, instead of on doing relevant work.

For our H.264 example from Sect. 3, the switching operation adjusts the supply voltage and processor frequency. Its overhead in time and energy depends on the implementation. Using the hardware circuit presented in [4] for switching, the overhead measured in time is up to 70 μs and in energy up to 4 μJ. These overheads affect both the final system cost (e.g., more energy consumption) and its run-time properties (e.g., more deadline misses because of time overhead). It is important to compare the time overhead with the minimum time the system stays in a scenario, which is equal to the required period between two consecutive frames (or smaller due to late scenario prediction). For a throughput of 30 frames per second, a switch may be acceptable between each pair of consecutive frames, as the overhead represents up to 0.2% of the time (70 μs out of 33 ms). On the other hand, for an application with, for example, 2500 RTSs per second, the switch overhead per frame represents 20% of the time, so the switches should be quite rare. Another illustration of the very small overhead of the run-time controller is reported in [82].

The way how the exploitation step encodes the scenarios into the system affects the switching cost. As we already mentioned, in the H.264 example, for each scenario a frequency-voltage pair is stored. However, for other exploitation examples, potentially a full copy of the source code for each scenario should be stored. These copies introduce large supplementary costs into the final system for each added scenario, and limit the total number of scenarios. The technique presented in our work can find a scenario set which achieves the best data memory optimization for the given instruction memory overhead. For a scenario that is rarely activated, its source code may be kept in a compressed version to reduce the storage cost, but as a decompression is done when the scenario is enabled, this increases the switching overhead.

Thus, the overhead for switching between two scenarios depends on what the run-time switching implies, and the scenarios between which the application switches. The switching overhead affects both the final system cost (e.g., more energy consumption) and its run-time properties (e.g., more deadline misses because of time overhead). In the design-time switching step, in parallel with deriving the switching mechanism, the set of scenarios, and consequently the predictor, may need to be adapted. This adaptation has to take into account the cost of each scenario, how often the switch between each pair of scenarios appears at run-time and how expensive it is. Two scenarios that were considered separately so far but which have a relatively close cost, and between which the system switches very often at run-time may need to be merged in a scenario with the worst-case cost among them.

The time overhead introduced by switching may cause undesired side-effects in a system, such as deadline misses in an H.264 decoder. Besides system and context dependent ways of handling such side-effects (e.g., an H.264 decoder may display the previous frame again if the deadline for the current frame is missed), we have looked at a general way for minimizing the side-effects that are caused by the time overhead introduced by the switching mechanism. The most conservative way to handle this overhead is to reserve time in each scenario, considering that the scenario is always activated for only one RTS in a row and taking into account the largest switching time that may appear. This approach can become very expensive, in which case it is a viable solution only for systems that require hard guarantees.

For systems where more freedom is acceptable, in each scenario, we may reserve time considering the switching time overhead averaged over the typical number of subsequent RTSs spent in a scenario, and the possible over-estimation in timing requirements that exist in the scenario. Such an over-estimation appears because for all RTSs clustered into a scenario, their worst-case cost is always considered when the scenario appears.

Moreover, some buffers are present already in almost all systems in our target domain, such as an output buffer in a video decoding system, which potentially can be used to compensate for the overhead variations that appear at run-time.

5.6 Calibration

The previous presented steps of our methodology make different design-time choices (e.g., scenario set, prediction algorithm) that depend very much on the possible values of RTS parameters, typically derived using profiling. This approach is obviously limited by our ability to predict the actual run-time environment, including the input data. For more dynamic systems, this will potentially lead to suboptimal run-time issues, like encountering an RTS that was not considered in the design-time choices, or an RTS with a higher cost than the one of the scenario to which it is predicted to belong. The first case appears when an RTS occurs at run-time of which the snapshot was not met during the identification step. In the second case, its snapshot was considered during the identification step, but the worst-case

cost observed for that snapshot is smaller than the actual cost of this RTS. This is also related to a possibly imperfect choice of the parameters or simplification of the predictor. Therefore, calibration can be used at run-time to complement the methodology steps previously presented. And even if RTS parameters guarantee correct prediction and costs are conservative, calibration can be useful to exploit, for example, a situation in which low-cost RTSs occur for long periods of time in a row.

To enable this, at run-time, information can be collected about actual values of the RTS parameters, the predicted scenario, the decisions taken by the switching mechanism, the measured cost for each scenario prediction, and the quality of the resulting system (e.g., the number of deadline misses). Both the execution cycles of the collecting process and the amount of stored information should be small as the collection is executed for each RTS. To keep the overhead limited, the calibration mechanism therefore should have access to only a limited amount of information. Moreover, it should be implemented as a low complexity algorithm.

Periodically, sporadically (e.g., when time slack is found into the system), or in critical exception situations (e.g., when the system quality is too low due to a certain number of missed deadlines), the calibration mechanism is then enabled. Based on the collected information, it may either change the ranges of parameter values and knob positions that characterize each scenario or adapt the scenario set by clustering existing scenarios or introducing new ones. In these cases, the prediction and maybe the switching mechanism have to be adapted as well. However, during the calibration, we have decided not to add new parameters or knobs, because this would lead to a too complicated and expensive process. Indeed, then to exploit the new parameters the predictor should be redesigned and for the new knobs the scenario exploitation step should be performed again, which is prohibitive at run-time.

Depending on the optimization applied in the exploitation step, the most common operations in the two abovementioned categories that can be performed efficiently considering the calibration's limited processing and storage budgets are:

1. To consider new RTSs that were not considered at design-time, and to map them to the scenario where they fit the best, based on the cost function, or to a new scenario. In this case, the predictor and the switching mechanism are also extended. As the complexity of the extension algorithm should be low, the resulting predictor will in general not be as efficient as if a new predictor were derived from scratch taking into account these new RTSs. Moreover, because an explosion in scenario storage has to be avoided, not for each RTS a new scenario can be created, but only for the ones which appear frequently enough to be promising for our final objective or problematic in terms of system quality.
2. To increase the actual cost of a scenario, based on its RTSs observed at run-time. This case may appear because the RTSs are defined using a limited set of parameters, and it is possible that multiple equivalent RTSs exist with different cost and only the cheaper ones were considered at design-time. The same problem may occur also when prediction quality is low, if many RTSs are

incorrectly predicted to belong to a scenario with a cost that is too low (under-prediction).
3. To increase the cost of some or all scenarios, because the run-time overhead introduced by related scenario mechanisms (e.g., prediction) is higher than anticipated. The same problem appears when the run-time overhead variations are too high and the buffering in the system cannot handle those variations. These cases are related to the fact that the input data and the environment in which the system runs is an extreme case (e.g., a lot of scenario switches), and the system was dimensioned for the average case.
4. To decrease the cost of a scenario, when only the RTSs with a low cost from that scenario actually appear at run-time. This improves our system cost (e.g., reducing energy), but potentially affects system quality negatively when RTSs with a higher cost occur. To maintain system quality, the cost can be increased again via the mechanism described in item two of this list. Alternatively, by monitoring the scenario, the scenario cost may be reset to the value that it had before calibration when one or a few of its RTSs with a higher measured cost than the current scenario cost occur.

All the operations presented above have the role to control and to guarantee the system quality, and to further improve our objective (i.e., to reduce the system cost) by exploiting the information collected at run-time. Our proposed implementation inserts in the final application some calibration code in line with the structure presented in Fig. 2.12. This code is executed immediately after each RTS was executed. While the information gathering (line 1) and the small adaptations (line 2) are executed for each RTS, the different calibration algorithms are executed periodically (lines 3–11) to limit the introduced overhead and to give a chance to the system to become stable between two consecutive calibrations. The small adaptations are low complexity algorithms which are enabled usually when either severe quality problems occur, and the adaptation cannot be delayed as the problems will really bother the end user, or when collecting and storing the information for a later calibration is more expensive than executing the calibration on the spot. Moreover,

```
Calibration (INT RTSCounter, ...)

1  InformationGathering()
2  SmallAdaptations()
3  for i= 1 to noCriticalCalibrations
4       do if ((RTSCounter - cCalib[i].LastActivation) > cCalib[i].Period)
5          then cCalib[i].FN()
6               cCalib[i].LastActivation - RTSCounter
7  for i=1 to noNonCriticalCalibrations
8       do if ((RTSCounter - nCalib[i].LastActivation) > nCalib[i].Period)
9          then if EnoughSlack(nCalib[i].WCEC)
10              then nCalib[i].FN()
11                   nCalib[i].LastActivation - RTSCounter
```

Fig. 2.12 Calibration structure

these adaptation algorithms usually update the currently selected scenario, while the calibration algorithms examine and calibrate all possible scenarios of the system.

To avoid introducing too much processing overhead in the processing of one RTS, each calibration algorithm has a different activation period. Moreover, the algorithms are divided into two categories, namely critical algorithms (lines 3–6) and non-critical algorithms (lines 7–11). The critical ones usually deal with the application constraints (e.g., deadlines or image quality), and are executed with an exact period. The non-critical ones deal with run-time tuning for cost reduction, and they can be postponed until enough slack remains after processing an RTS, such that their execution will certainly not result in a quality degradation.

For our H.264 example from Sect. 3, a non-critical periodic calibration creates new scenarios when so far unseen frame macro-block breakups appear at run-time, and for each scenario sets the frequency-voltage pair based on the measured computation cycles required to decode that frame. When a given maximum number of scenarios has been reached, each time a new scenario needs to be introduced, one of the earlier introduced scenarios is removed again (using some heuristic to select that scenario). Moreover, another non-critical periodic calibration checks for the case when a large difference in computation cycles appears between the amount reserved for a scenario and the measured amount for all RTSs characterized to be in that scenario, so the frequency-voltage pair is modified to reduce the system energy consumption. However, when this adapted scenario introduces missed deadlines, the pair is reset to its initial value by a small adaption that is run for each RTS.

Before we go to the case studies illustrating this generic approach, we will also show the more generic flow for the data variable system scenario variant. That one is illustrated in Fig. 2.13. The main differences are in the way the identification, prediction/detection, and switching happen. They are partitioning based and not clustering based as already indicated in Sect. 4. In this flow also the impact of combining this with a task concurrency management methodology (see [43]) is incorporated. That is especially important when mapping multiple threads on a single processor or many tasks to a multi-processor platform (see also Sect. 7).

6 Case Study Summary

In this section, we present a summary of some major case studies we have performed and which substantiate our claims on the broad applicability, the efficiency, and cost saving potential of our system scenario approach. Several of these are also further refined in the subsequent application chapters of this book. Besides the already presented H.264 case study in Sect. 3, in our research groups other approaches have been developed at various levels of design abstraction, which are relevant and demonstrate the proposed scenario-based methodology from many different angles.

The scenario concept was first used in the context of multi-task applications in [45, 85] to capture the data-dependent dynamic behavior inside a thread, and hence to better schedule a multi-thread application on a heterogeneous multi-processor

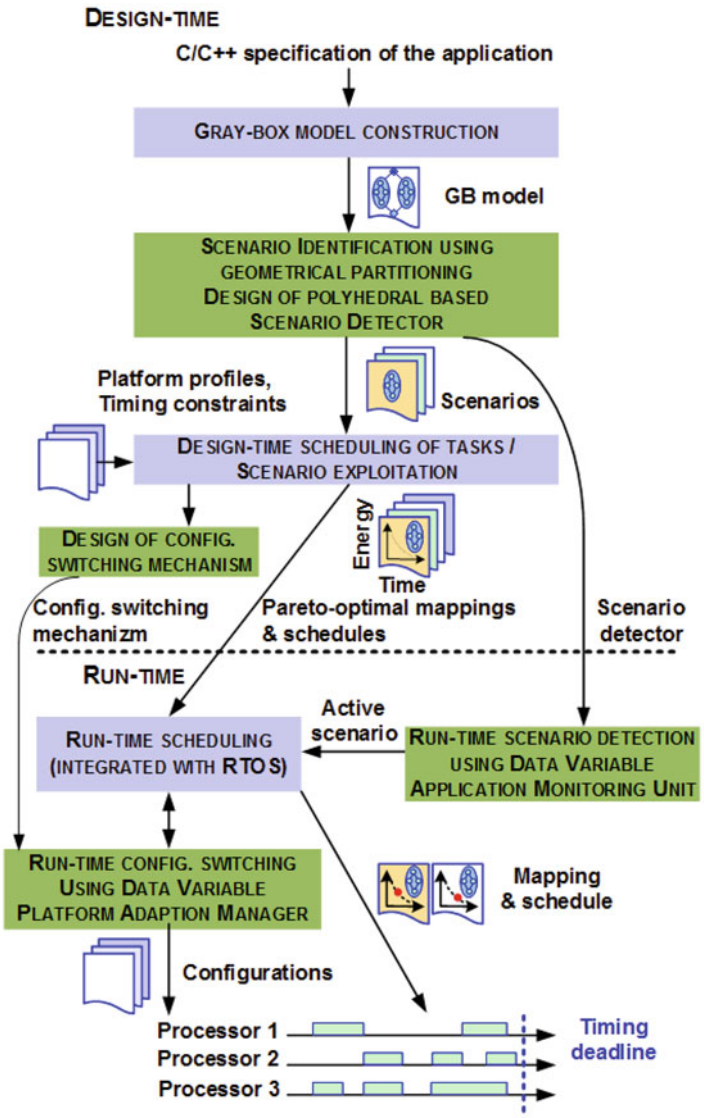


Fig. 2.13 Overall flow of data variable system scenario methodology

architecture, allowing the change of voltage level for each individual processor. That work also includes a system-scenario-based DVFS hybrid design-time/run-time scheduler technique. However, the scenario identification and run-time predictor are done manually.

In [28], we have presented a general toolflow that can be applied to streaming applications to reduce their energy consumption by taking advantage of the DVFS

mechanism available in modern processors. In this work, we aim at applications that are often implemented as a main loop, called the loop of interest, that is executed over and over again, reading, processing, and writing out individual stream objects (e.g., the H.264 video decoder presented in Fig. 2.4). Each iteration of the loop of interest has a time constraint due to the throughput required by the system (usually specified by a standard). To identify the RTS parameters, our toolflow uses the techniques presented in Sect. 5.2. The scenarios are derived by clustering the identified RTSs using as a cost function the required cycle budget (see also Sect. 5.2). The flow has been applied both to the H.264 video decoder benchmark which has already been discussed in this summary chapter (see Sect. 3 and to an MP3 audio decoder. For the latter, the best obtained result contains 17 scenarios. Then, the energy reduction obtained is 16% for stereo and 24% for a mixed set of stereo and mono streams, respectively, for a miss ratio of up to one frame per 3 min (0.013%).

In [49, 50], we have presented an instantiation of our scenario methodology in the form of a general toolflow that can be applied to streaming applications to reduce their energy consumption by applying the scenario concepts on top of the data transfer and storage exploration (DTSE) methodology, particularly to enable more loop transformations in the DTSE flow. The target application domain contains data-dominated regular streaming applications. We can obtain a significant additional gain (of nearly a factor 2) on top of the application of only the DTSE flow on the MP3 audio decoder.

In [44], the scenario concept has been merged into a dynamic memory management methodology intended for embedded systems. It has been demonstrated for optimizing dynamic memory allocation (i.e., malloc()/free()) for the IPv4 layer in an IEEE 802.11b wireless network application. This has led to the design of a customized dynamic memory allocator where we manage to achieve on average reduction of 90% for the energy consumption, of 62% for memory footprint, and an 81% improvement of performance compared to the dynamic memory allocator used in the Linux operating system.

In another study [76], we have used the scenario approach to adapt the hardware configuration of high performance processors for energy efficiency. When a program is being executed on a high performance processor, it typically does not need all available processor resources at the same time. The amount of resources that a program needs actually heavily depends on the code that is being executed and typically varies over time [61]. By switching off the processor resources that are not needed during the execution of a program, the energy consumption of the processor can be reduced substantially without slowing down the execution of the program.

In [59], the energy and performance efficiency of memories is increased using scenario-based methodologies that tackle process variability problems. The paper shows that apart from the application RTSs, another significant source of unpredictability is the platform itself when using technology scaling beyond the 90nm technology node. This approach has been evaluated on an aggressive 8-issue out-of-order superscalar processor microarchitecture with an adaptive branch predictor, processor width, re-order buffer, fetch buffer, and caches. On average, our hardware

adaptation approach achieves a 36% reduction in energy consumption with a performance penalty of only 2.9% (instructions per cycle).

Another memory organization oriented body of work has focussed on dealing with L1 data cache mapping of dynamic wavelet-based video codecs (see [16, 17]). As already indicated in Sect. 4, quite complex but run-time efficient parametrized equations have been derived to combine many different Pareto curve working points and positions in compact representations which are tightly characterizing the actual run-time situations. The case of even more strongly dynamic video codecs has been dealt with in [18].

In [3], it is illustrated that the design of wireless protocol systems benefits from the scenario concept by characterizing wireless channels and user conditions as RTS parameters. The predicted scenarios are used to exploit cross-layer (in the OSI model) trade-offs and switch between knob positions, adjusting the user throughput and system power accordingly at run-time. Also in the same wireless protocol domain, the work presented in [54] manages to reduce energy consumption considerably by exploiting run-time controllable knobs of actual RF components and the 802.11 medium access controller.

In [53], a methodology is introduced that identifies scenarios of wireless mobile streams in multi-threaded applications. The RTSs consider the change in size among packets in a wireless network stream. The identified scenarios are then used to control at run-time a knob that switches DMA block transfers of heap data on and off and thus it saves energy and execution cycles accordingly.

In [81, 83, 84], the scenario concept is used for energy-efficient scheduling of tasks for heterogeneous MPSoC embedded platforms. The scenarios are extracted from the task graph of the software applications and used to reduce energy through the proposed DVFS methodology.

In [55], a method for estimating the execution time of stream-oriented applications mapped on a multi-processor on-chip is detailed. For this type of systems, the pipelined decoding of sequential streaming objects has a high impact on achieving the required throughput. The application is modeled as a homogenous synchronous dataflow graph (HSDF). Within the application's loop of interest the scenarios are manually defined based on the different execution workloads of tasks. An accurate execution time estimation method is proposed that supports parallel and pipelined decoding of streaming objects, taking into account the transient and periodic behavior of scenario executions and the effect of scenario transitions.

A more general model than HSDF that is still analyzable is the scenario-aware dataflow model (SADF) [74]. It is a design-time analyzable stochastic generalization of the synchronous dataflow (SDF) model, which can capture several dynamic aspects of modern streaming applications by incorporating system scenarios. The scenarios and the run-time predictor are explicitly described in the model. So no further need is present for identification of scenarios for applications written using this model. Moreover, analysis of long-run average and worst-case performance are decidable. SADF combines both analyzability and explicit representation of scenarios. The only current drawback is that not all possible forms of dynamism (e.g., interactions with external events) can be represented with it.

Chapter 8 provides a good overview of the current status of this work and the extensions that have been added compared to the initial version of [74]. In particular, parametrization of actor rates and execution times has been added to the SADFs, leading to a broader applicability in more dynamic applications [64]. That has enabled worst-case throughput analysis [65], worst-case latency analysis [67, 70], and fully parametrized dataflow scenarios [66, 71]. For the important subdomain of streaming applications, more efficient formalizations have been derived [63]. A further extension has included the link to timed automata [68], including the translation from FSM-SADF models [69].

The DVFS knob used very often throughout our work usually assumes that timing constraints are at least partly soft. As already mentioned, soft requirements allow to trade off quality and cost. In [25], we illustrate a DVFS application that provides hard guarantees. Scenario identification is based entirely on a conservative static analysis. Also the method of [55] described above is suitable to provide hard guarantees. In general, it may not always be easy to make all steps of the scenario methodology conservative, but whenever this is possible our methodology applies, omitting calibration which is not meaningful in a context requiring hard guarantees. This has also been extended further for the data variable case in [86, 87, 89], using especially the H264/AVC video codec application as main driver. The DVFS knob has also been combined with a RTH (run-to-halt) platform knob to achieve even larger gains [89]. Chapters 4 and 5 provide a good overview of the current status of this DVFS related work and the extensions that have been added compared to the initial version of [25].

At a higher design abstraction level, in [10], scenarios for memory management of the heap data are defined by the user using the steady-state concept. The identified scenarios are used to switch between dynamic data types according to the predicted scenarios in multimedia software applications. In [73], the design abstraction level is raised even higher and scenario-based optimizations are made at the modeling level. In this last paper, scenarios are identified based on behavioral information and different UML transformations are proposed for the abstract data types of multimedia applications in order to achieve trade-offs between energy, memory accesses, and memory footprint.

Also situated in this context of memory organization, a memory-aware system scenario approach has been developed [20]. It has been applied for the energy-efficient optimization of dynamic multi-media applications [21]. And it has also been combined with a systematic approach to perform data interleaving [22].

A variant of the scenario concept with a much bigger number of RTSs to accommodate even more dynamic situations has been used in [72] for 3D graphics applications. It considers so-called sub-scenarios that exhibit specific correlated behavior and that are characterized individually at design-time, but that are merged only at run-time. This allows to break up very large systems where the number of scenarios would otherwise explode. This is a very powerful extension which will not be further covered in the scope of this book though.

To illustrate that the system scenario concept can also be expanded to domains far beyond digital systems, we refer to the recent work we have produced in

photovoltaics modules. In order to perform scalable modeling of the complex electrical-thermal-optical effects affecting such modules, and this under the presence of non-uniform irradiation conditions such as due to static and dynamic shading, the more conventional modeling approach has been extended with "spatial system scenario's" in [2]. And we are also studying "temporal system scenario's" as an extension of the data variable techniques, for renewable energy forecasting purposes.

7 Extension to Multi-Tasking and Multi-Threading on Multi-Processor Platforms

Although some of the surveyed applications of the system scenario concepts target multi-task or multi-processor systems, the use of scenarios within a multi-processor embedded system design trajectory has not been extensively explored yet. Given the importance of multi-processor systems, we consider this as an interesting direction for future research. A design flow like the one sketched in Fig. 2.14 can be envisioned. The flow starts with extracting intra-task scenarios for each application task, and based on them derives the inter-task application scenarios. The intra- and inter-task scenarios are conceptually the same from a methodology perspective, so that RTS parameter discovery techniques and scenario prediction techniques can be reused. However, intra- and inter-task scenarios have a different impact on the intra- and inter-task parts of the remaining design trajectory, their exploitation being in general quite different.

Certain parts of the scenario methodology will be affected. Characterizing the cost (e.g., the cycle budget) of an RTS, which is part of scenario identification, has to be adapted to accommodate the specific problems that appear in multi-task applications, like intra- and/or inter-processor scheduling, inter-task communication costs and delays, pipelined execution. These problems make the resource estimation for multi-task applications, especially in a multi-processor context, a challenging research topic. The accuracy of resource estimation does not only affect scenario identification, but also calibration. It may furthermore be necessary to distribute calibration or to collect calibration information at a central location in the system. After deriving inter-task system scenarios, they are used in decision making along the design trajectory, like in task binding and scheduling. Depending on the type of exploitation, various switching mechanisms may be needed, varying from configuring simple settings to migrating tasks from one processor to another one. Advanced switching mechanisms such as run-time task migration are another interesting research direction. Finally, if multiple scenario-aware applications can coexist in the same multi-application system, scenario-aware resource and quality of service management across applications needs to be investigated.

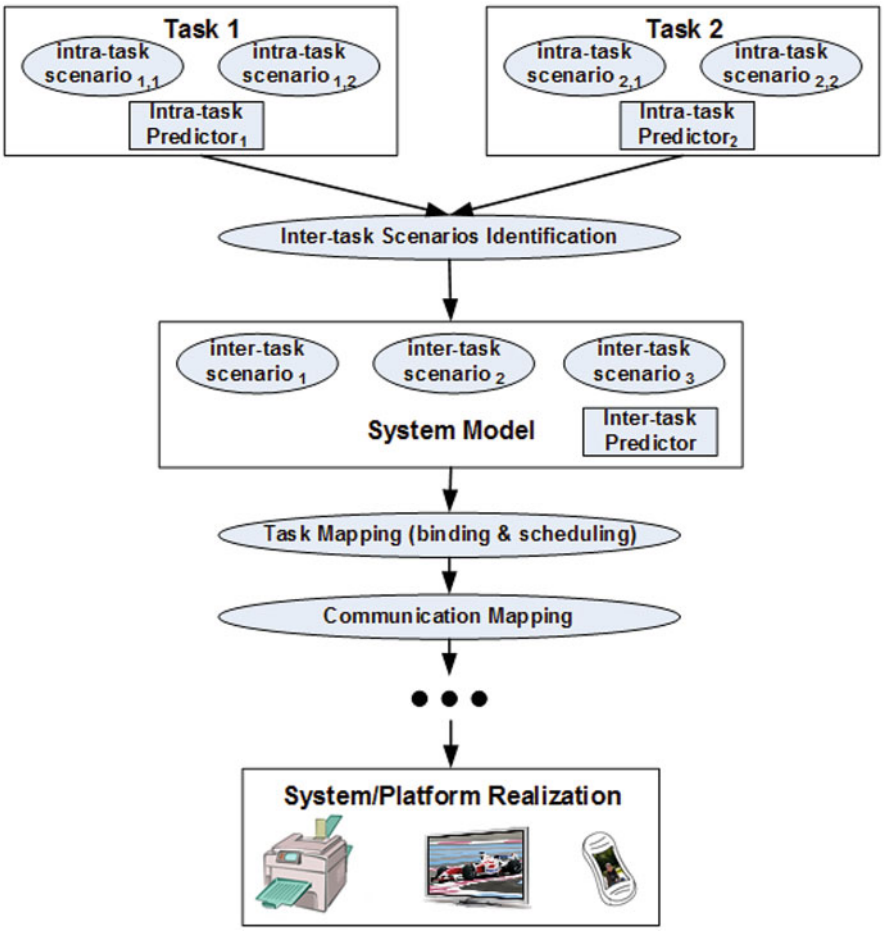


Fig. 2.14 Extended design flow for multi-task multi-processor systems

8 Related Work

This section consists of two parts. The first part compares our system-scenario-based methodology with related approaches, while the second part presents exploitation examples of scenarios found in the literature (not related to our own efforts, which were already surveyed in the previous sections).

8.1 Related Design Approaches

In the past, embedded system design was significantly improved using the inspector-executor technique, which has been developed at the University of Maryland in the

early 1990s [58]. The basic idea behind it is to compile the application loops in two phases, an inspector and an executor. The inspector examines the data access pattern in the loop body and creates a schedule for fetching the values stored in remote memories. The executor retrieves remote values according to the schedule and executes the loop. The authors have studied run-time methods to automatically parallelize and schedule iterations of a loop in certain cases when compile-time information is inadequate. At compile-time, these methods set up the framework for performing a loop dependency analysis. At run-time, wavefronts of concurrently executable loop iterations are identified and the loop iterations are reordered for increased parallelism.

A similar approach has been taken also in [1] where a loop with irregular assignment computations contains loop-carried output data dependencies that can only be detected at run-time. A load-balanced method based on the inspector-executor model is proposed to parallelize this loop pattern. The basic idea lies in splitting the iteration space of the sequential loop into sets of conflict-free iterations that can be executed concurrently on different processors.

In [92], the authors propose a modified inspector-executor method for implementing accesses to a distributed array. In the method, the compiler runs an inspector during compile time to obtain the information of data dependencies among node processors, and it uses that information to optimize communication code included in the executor.

In [77], a novel strategy is discussed, which dynamically drives the communication between the processors by examining the content of the data at run-time in order to reduce communication costs for numerical weather prediction modes. Compared to the inspector-executor which is based on low-level data access patterns, this strategy includes high-level application dependent information.

System workload characterization is another related field of research. It is particularly relevant for the scenario identification step of our methodology. It has gained interest already more than 30 years ago [19]. First, it has been used for selecting the appropriate workload for doing meaningful measurements on the performance of computer systems. Later, workload characterization has been extended to wired [41] and wireless [40] networks. Moreover, it also was considered as a base for traffic shaping which is used for adapting the workload to the expected workload in the network/application [56].

A specific area in workload characterization is the identification of program phases [61]. Programs usually consist of a number of repeating execution patterns, which are identified. In the program phase detection, code-based phase detection techniques [35] and interval-based phase detection techniques [61] are used. In code-based phase detection, program phases are associated with functions and loops. The interval-based phase detection techniques divide the execution of a program into fixed-length instruction intervals and group intervals with similar characteristics.

A detailed survey about workload characterization can be found in [5]. It identifies five common steps followed by all workload characterization approaches, including our scenario identification techniques: (1) choice of the set of parameters

able to describe the behavior of the workload, (2) choice of the suitable instrumentation, (3) experimental measurement collection, (4) analysis of the workload data, and (5) construction of workload models.

Workload characterization and the inspector-executor technique perform most of the analysis at run-time. This approach is beneficial when design-time analysis is not available. The system scenario methodology for designing embedded systems is more general in the sense that it can handle systems with unpredictable and extremely varying workloads where the previous techniques cannot be used. The system is made more predictable via design-time analysis. The actual behavior of the system, obtained by combining static analysis and profiling approaches, is split into distinct classes (scenarios) of typical workload behavior. System scenarios allow optimization of the system mapping for each scenario, optimizations from which the system profits when the scenario appears at run-time. This combination of design-time analysis and classification of behaviors with run-time exploitation and potentially calibration is the main novelty of the scenario-based approach.

Due to the presence of the run-time calibration step in our methodology, the scenario approach is related to adaptive controllers [15]. However, the scenario approach distinguishes itself via the design-time preparation and classification of system behaviors, which guides the calibration into the most promising directions (by pruning directions that are known to be of no interest). Furthermore, for cost reasons, at run-time, our calibration technique is only active at certain designated moments in time (calibration time) whereas a typical adaptive controller executes continuously.

The system scenario concept was identified explicitly for the first time in [85], where it was used to improve the mapping of dynamic applications onto a multiprocessor platform. Concepts closely related to the scenario idea already appear in [45]. Our control variable oriented scenario methodology ideas were for the first time briefly introduced in [26]. That paper does not detail any of the methodology steps, and even entirely omits the switching and calibration steps.

8.2 Scenario Exploitation Examples in Literature

Scenario-like concepts were applied in an ad-hoc manner several times in the literature, with an emphasis on exploiting scenarios, and not on systematically identifying and predicting them.

In [9], the authors use in a systematic way the information about periodicity of multimedia applications to present a new concept of DVFS. Each period in the application shows a large variation in terms of execution time. The proposed idea is to supply the information of the execution time variations in addition to the content itself. This makes it possible to perform DVFS independent of worst-case execution time estimation providing energy consumption reduction of client systems compared to previous DVFS techniques. However, the authors do not specify how the periods should be identified.

In [60], for each manually identified scenario, the authors select the most energy efficient architecture configuration that can be used to meet the timing constraints. The architecture has a single processor with reconfigurable components (e.g., number and type of function units), and its supply voltage can be changed. It is not clear how scenarios are predicted at run-time.

In [8], a reactive predictor is used to select the lowest supply voltage for which the timing constraints of an MPEG decoder are met. An extension [57] considers two simultaneous resources for scenario characterization. It looks for the most energy-efficient configuration for encoding video on a mobile platform, exploring the trade-off between computation and compression efficiency.

In the context of multi-task applications, in [46], scenarios are characterized by different communication requirements (such as different bandwidth, latency constraints) and traffic patterns. The paper presents a method to map an application to a network on chip (NoC) architecture, satisfying the design constraints of each individual scenario. This approach concentrates on the communication aspect of the application mapping. It allows dynamic network reconfiguration across different scenarios. As the over-estimation of the worst-case communication is very large, this method performs poorly on systems where the traffic characteristics of scenarios are very different or when the number of scenarios is large. In [47], the method has been extended to work for these cases too.

In [42], a combination of a hierarchical finite state machine (FSM) with a synchronous dataflow model (SDF) is used to capture scenarios within a multi-task streaming application. The FSM represents the scenarios' run-time detector. The scenarios are identified by the designer and they are already described in the model. The authors have shown that by writing the application in this model, the scenario knowledge can be used to save energy when mapping the application on one processor. The SADF model of computation of [74] generalizes this FSM-SDF model of computation and it includes the formalization of using system scenarios. As summarized in Sect. 6, this allows a systematic exploitation of system scenarios in the SADF and HSDF contexts.

Another example of improving a multi-task application analysis approach using application scenarios is [79]. This paper extends an existing method for performance analysis of hard-real time systems based on real-time calculus, taking into account correlations that appear between different components of the system. The knowledge about these correlations is used to derive the system scenarios. The authors present only how these scenarios could be modeled in their high-level modeling/analytical approach, but no way to identify scenarios and no prediction mechanism was considered.

As a final observation, quality of service (QoS) mechanisms may make use of scenarios and related concepts. However, a detailed discussion of QoS techniques is beyond the scope of this book. More information can be found in papers related to QoS, like [30, 78].

In summary, compared to the previous work on workload characterization and on the inspector-executor approach, which target a purely run-time approach, the methodology advocated in this book targets a combined design-time and run-time

approach with specific characteristics. The early work in system scenarios is either quite ad-hoc, e.g., [9, 47, 60], or targets specific contexts, e.g., [28, 31, 50, 85]. Only since our paper in [29], the systematic design flow has been consolidated for systems based on control variables. And only more recently also the data variable option has been added [33, 87].

9 Conclusions

In this chapter, we have introduced the basic concept and methodology steps of the system scenario approach. This approach clusters the run-time situations (RTSs) in which a system may run based on similarities from a multi-dimensional cost trade-off perspective (e.g., resource utilization), in such a way that a system can be optimized at design-time and configured at run-time to effectively exploit this cost similarity.

Different from the well-known use-case scenarios, which are manually written diagrams that represent the user perspective on future system usage, system scenarios can be automatically derived, are transparent to the user, and focus on system cost optimization.

A combined design-time/run-time methodology for using system scenarios to improve the final system cost is detailed with 5 distinct steps which are each further refined.

Furthermore, several case studies are briefly summarized to show how the methodology can be applied to a broad range of real life design problems. The diversity of these case studies, and the other given exploitation examples, emphasizes the fact that, although the various design problems look different, they can be covered by the same underlying methodology. The obtained reductions in the final system costs prove that applying our methodology in the design process leads to better products.

References

1. M. Arenaz, J. Tourino, R. Doallo, An inspector-executor algorithm for irregular assignment parallelization, in *2nd International Symposium on Parallel and Distributed Processing and Applications (ISPA 2004), Hong Kong* (2004), pp. 4–15
2. M. Baka, F. Catthoor, D. Soudris, Proposed evaluation framework for exploration of smart PV module topologies, in *Proceedings of European Photovoltaic Solar Energy Conference (PVSEC)*, poster presentation 1BV.5.36, Munich (2016), pp. 176–179
3. B. Bougard, S. Pollin, F. Catthoor, W. Dehaene, Cross-layer power management in wireless networks and consequences on system-level architecture. Signal Process. J. **86**(8), 1792–1803 (2006)
4. T. Burd, T. Pering, A. Stratakos, R. Brodersen, A dynamic voltage scaled microprocessor system. IEEE J. Solid-State Circuits **SC-35**(11), 1571–1580 (2000)

5. M. Calzarossa, G. Serazzi, Workload characterization: a survey. Proc. IEEE **81**(8), 1136–1150 (1993)
6. J.M. Carroll, (ed.), *Scenario-Based Design: Envisioning Work and Technology in System Development* (Wiley, New York, 1995)
7. F. Catthoor, (ed.), *Unified Low-Power Design Flow for Data-Dominated Multi-Media and Telecom Applications* (Kluwer Academic Publishers, Boston, 2000). ISBN 0-7923-7947-0
8. K. Choi, K. Dantu, W.-C. Cheng, M. Pedram, Frame-based dynamic voltage and frequency scaling for a MPEG decoder, in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (ACM Press, New York, 2002), pp. 732–737
9. E.-Y. Chung, L. Benini, G. De Micheli, Contents provider-assisted dynamic voltage scaling for low energy multimedia applications, in *Proceedings of IEEE International Symposium on Low Power Design (ISLPED), Monterey* (2002), pp. 42–47
10. E.G. Daylight, S. Wuytack, C. Ykman-Couvreur, F. Catthoor, Analyzing energy friendly steady state phases of dynamic application execution in terms of sparse data structures, in *Proceedings of IEEE International Symposium on Low Power Design (ISLPED), Monterey* (2002), pp. 76–79
11. S. Debray, W. Evans, R. Muth, B. De Sutter, Compiler techniques for code compaction. ACM Trans. Program. Lang. Syst. **22**(2), 378–415 (2002)
12. V. Desmet, H. Vandierendonck, K. De Bosschere, 2far: a 2bcgskew predictor fused by an alloyed redundant history skewed perceptron branch predictor. J. Instruction-Level Parallelism **7**, 1–11 (2005)
13. B. De Sutter, B. De Bus, K. De Bosschere, Link-time binary rewriting techniques for program compaction. ACM Trans. Program. Lang. Syst. **27**(5), 882–945 (2006)
14. B.P. Douglass, *Real Time UML: Advances in the UML for Real-Time Systems* (Addison Wesley, Reading, 2004)
15. G. Dumont, M. Huzmezan, Concepts, methods and techniques in adaptive control. Proc. Am. Control Conf. **2**, 1137–1150 (2002)
16. V. Ferentinos, M. Milia, G. Lafruit, J. Bormans, F. Catthoor, Memory compaction and power optimization for wavelet-based coders, in *Proceedings of IEEE Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), Torino* (2003), pp. 328–337
17. V. Ferentinos, G. Lafruit, M. Milia, J. Bormans, F. Catthoor, T. Stouraitis, Optimized memory requirements for wavelet-based scalable multi-media codecs. J. Embed. Comput. **1**(3), 363–380 (2005)
18. V. Ferentinos, B. Geelen, F. Catthoor, G. Lafruit, T. Stouraitis, R. Lauwereins, D. Verkest, Adaptive mapping to resource availability for dynamic wavelet-based applications, in *Proceedings of IEEE 5th Estemedia Workshop (ESTIMEDIA), Salzburg* (2007), pp. 53–58.
19. D. Ferrari, Workload characterization and selection in computer performance measurement. IEEE Comput. **5**(4), 18–24 (1972)
20. I. Filippopoulos, P.-G. Kjeldsberg, E. Hammari, F. Catthoor, J. Huisken, Memory-aware system scenario approach energy impact, in *Proceedings of 30th Norchip Conference, Copenhagen* (2012)
21. I. Filippopoulos, P.-G. Kjeldsberg, E. Hammari, F. Catthoor, J. Huisken, Exploration of energy efficient memory organisations for dynamic multimedia applications using system scenarios, in *MeAOW Workshop, Montreal* (2013)
22. I. Filippopoulos, N. Sharma, F. Catthoor, P.-G. Kjeldsberg, P. Panda, Integrated exploration methodology for data interleaving and data-to-memory mapping on SIMD architectures. ACM Trans. Embed. Comput. Syst. **15**(3), 59 (2016)
23. M. Fowler, Use cases, in *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd edn., chap. 9 (Addison Wesley, Reading, 2003), pp. 99–106
24. M.C.W. Geilen, T. Basten, B.D. Theelen, R.H.J.M. Otten, An algebra of Pareto points. Fund. Inform. **78**(1), 35–74 (2007)
25. V. Gheorghita, T. Basten, H. Corporaal, Intra-task scenario-aware voltage scheduling, in *Proceedings of IEEE International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)* (2005), pp. 177–184

26. V. Gheorghita, T. Basten, H. Corporaal, Application scenarios in streaming-oriented embedded system design, in *Proceedings of IEEE International Symposium on System-on-Chip (SoC 2006), Tampere* (2006), pp. 175–178
27. V. Gheorghita, S. Stuijk, T. Basten, H. Corporaal, Automatic scenario detection for improved WCET estimation, in *Proceedings of 45th ACM/IEEE Design Automation Conference (DAC), San Francisco* (2008), pp. 101–104
28. V. Gheorghita, T. Basten, H. Corporaal, Scenario selection and prediction for DVS-aware scheduling. J. Signal Process. Syst. **50**(2), 137–161 (2008)
29. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, K. De Bosschere, System scenario based design of dynamic embedded systems. ACM Trans. Des. Autom. Embed. Syst. **14**(1), article 3 (2009)
30. K. Goossens, J. Dielissen, J. van Meerbergen, P. Poplavko, A. Radulescu, E. Rijpkema, E. Waterlander, P. Wielage, Guaranteeing the quality of services in networks on chip, in *Networks on Chip*, chap. 4 (Kluwer Academic Publishers, Hingham, 2003), pp. 61–82
31. J. Hamers, L. Eeckhout, K. De Bosschere, Exploiting video stream similarity for energy-efficient decoding, in *Proceedings of the 13th International Multimedia Modeling Conference (MMM)*. Lecture Notes in Computer Science, vol. 4352 (Springer, Berlin, 2007), pp. 11–22
32. E. Hammari, F. Catthoor, J. Huisken, P.G. Kjeldsberg, Application of medium-grain multiprocessor mapping methodology to epileptic seizure predictor, in *Proceedings of IEEE Norchip Conference, Tampere* (2010)
33. E. Hammari, F. Catthoor, P.G. Kjeldsberg, J. Huisken, K. Tsakalis, L. Iassemidis, Identifying data-dependent system scenarios in a dynamic embedded system, in *Proceedings of IEEE Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas* (2012), pp. 70–76
34. A. Hansson, M. Coenen, K. Goossens, Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip, in *Proceedings of Design, Automation, and Test in Europe Conference (DATE)* (IEEE Press, Piscataway, 2007), pp. 954–959
35. M. Huang, J. Renau, J. Torrellas, Positional adaptation of processors: application to energy reduction, in *Proceedings of International Symposium on Computer Architecture (ISCA), San Diego* (2003)
36. C.J. Hughes, J. Srinivasan, S.V. Adve, Saving energy with architectural and frequency adaptations for multimedia applications, in *Proceedings of 34th Annual International Symposium on Microarchitecture (MICRO-34)* (IEEE Computer Society, Washington, 2001), pp. 250–261
37. IEEE Standard 1471, Recommended practice for architectural description of software-intensive systems (2000)
38. M.T. Ionita, Scenario-based system architecting: a systematic approach to developing future-proof system architectures. Ph.D. Thesis, Technische Universiteit Eindhoven (2005)
39. N. Jha, Low power system scheduling and synthesis, in *Proceedings IEEE International Conference on Computer-Aided Design, San Jose* (2001), pp. 259–263
40. D. Kotz, K. Essien, Analysis of a campus-wide wireless network. Wirel. Netw. **11**(1), 115–133 (2005)
41. R. Lee, An introduction to workload characterization (1991). http://support.novell.com/techcenter/articles/ana19910503.html
42. S. Lee, S. Yoo, K. Choi, An intra-task dynamic voltage scaling method for SoC design with hierarchical FSM and synchronous dataflow model, in *Proceedings of IEEE International Symposium on Low Power Design (ISLPED), Monterey* (2002), pp. 84–87
43. Z. Ma, P. Marchal, D. Scarpazza, P. Yang, C. Wong, I. Gomez, S. Himpe, C. Ykman, F. Catthoor, *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogeneous Platforms* (Springer, Heidelberg, 2007). ISBN 978-1-4020-6328-2
44. S. Mamagkakis, F. Catthoor, D. Soudris, Middleware design optimisation of wireless protocols based on the exploitation of dynamic input patterns, in *Proceedings of 10th ACM/IEEE Design and Test in Europe Conference (DATE), Nice* (2007), pp. 1036–1041

45. P. Marchal, C. Wong, A. Prayati, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, H. De Man, Dynamic memory oriented transformations in the MPEG4 IM1-player on a low power platform, in *Proceedings of International Workshop on Power Aware Computing Systems (PACS), Cambridge* (2000), pp. 31–40
46. S. Murali, M. Coenen, A. Radulescu, K. Goossens, G. De Micheli, Mapping and configuration methods for multi-use-case networks on chips, in *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)* (ACM Press, New York, 2006), pp. 146–151
47. S. Murali, M. Coenen, A. Radulescu, K. Goossens, G. De Micheli, A methodology for mapping multiple use-cases onto networks on chips. *Proceedings of Design, Automation, and Test in Europe Conference (DATE)* (IEEE Press, Piscataway, 2006), pp. 118–123
48. T. Okabe, Y. Jin, B. Sendhoff, A critical survey of performance indices for multi-objective optimisation, in *Proceedings of the Congress on Evolutionary Computation*, vol. 2 (IEEE Press, Piscataway, 2003), pp. 878–885
49. M. Palkovic, H. Corporaal, F. Catthoor, Global memory optimisation for embedded systems allowed by code duplication, in *9th International Workshop on Software and Compilers for Embedded Systems (SCOPES), Dallas* (2005), pp. 72–79
50. M. Palkovic, H. Corporaal, F. Catthoor, Dealing with variable trip count loops in system level exploration, in *International Workshop on Optimizations for DSP and Embedded Systems (ODES), in Conjunction with International Symposium on Code Generation and Optimization, Manhattan* (2006), pp. 19–28
51. V. Pareto, *Manual of Political Economy (1906)* (translation of the 1927 edition). (A.M. Kelley, New York, 1971)
52. J.M. Paul, D.E. Thomas, A. Bobrek, Scenario-oriented design for single-chip heterogeneous multiprocessors. IEEE Trans.Very Large Scale Integr. Syst. **14**(8), 868–880 (2006)
53. M. Peon-Quiros, A. Bartzas, S. Mamagkakis, F. Catthoor, J. Mendias, D. Soudris, Direct memory access optimisation in wireless terminals for reduced memory latency and energy consumption, in *Proceedings of IEEE Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Lecture Notes in Computer Science, vol. 4644 Goteborg (2007), pp. 373–383
54. S. Pollin, R. Mangharam, B. Bougard, L. Van der Perre, I. Moerman, R. Rajkumar, F. Catthoor, MEERA: cross-layer methodology for energy-efficient resource allocation for wireless networks. IEEE Trans. Wirel. Commun. **56**(5), 606–621 (2007)
55. P. Poplavko, T. Basten, J.L. van Meerbergen, Execution-time prediction for dynamic streaming applications with task-level parallelism, in *Proceedings of 10th EuroMicro Conference in Digital System Design (DSD)* (IEEE Computer Society Press, Washington, 2007), pp. 228–235
56. B. Raman, S. Chakraborty, Application-specific workload shaping in multimedia-enabled personal mobile devices, in *Proceedings of the 4th International Conference on Hardware Software Codesign, Seoul* (2006), pp. 4–9
57. D.G. Sachs, S.V. Adve, D.L. Jones, Cross-layer adaptive video coding to reduce energy on general-purpose processors, in *Proceedings of IEEE International Conference on Image Processing* (IEEE Press, Piscataway, 2003), pp. 109–112
58. J.H. Saltz, R. Mirchandaney, K. Crowley, Run-time parallelization and scheduling of loops. IEEE Trans. Comput. **40**(5), 603–612 (1991)
59. C. Sanz Pineda, A. Papanikolaou, M. Prieto, M. Miranda, F. Catthoor, System-level process variability compensation on memory organisations of dynamic applications: a case study, in *Proceedings of IEEE International Symposium on Quality Electronic Design (ISQED), San Jose* (2006), pp. 376–382
60. R. Sasanka, C.J. Hughes, S.V. Adve, Joint local and global hardware adaptations for energy. ACM SIGARCH Comput. Architect. News **30**(5), 144–155 (2002)
61. T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, New York* (2002), pp. 45–57

62. D. Shin, J. Kim, Optimizing intra-task voltage scheduling using data flow analysis, in *Proceedings of the 10th Asia and South Pacific Design Automation Conference (ASPDAC)* (ACM Press, New York, 2005), pp. 703–708
63. M. Skelin, Worst-case performance analysis of scenario-aware real-time streaming applications. Doctoral Dissertation, N.Technical Univ. Norway, Trondheim and K.U. Leuven, Belgium (dual PhD) (2016)
64. M. Skelin, M. Geilen, F. Catthoor, S. Hendseth, Worst-case throughput analysis for parametric rate and parametric actor execution time scenario-aware dataflow graphs, in *1st International Workshop on Synthesis of Continuous Parameters (SynCop), Grenoble* (2014), pp. 65–79
65. M. Skelin, F. Catthoor, S. Hendseth, Worst-case throughput analysis of SDF-based parametrized dataflow models, in *Proceedings of Euromicro Symposium on Digital System Design (DSD), Funchal* (2015), pp. 17–24
66. M. Skelin, M. Geilen, F. Catthoor, S. Hendseth, Parametrized dataflow scenarios, in *Proceedings of 6th ACM/IEEE International Conference on Embedded Software (EMSOFT, Amsterdam* (2015), pp. 95–104
67. M. Skelin, F. Catthoor, S. Hendseth, Worst-case latency analysis of SDF-based parametrized dataflow models, in *Proceedings of Conference on Design and Architecture for Signal and Image Proceedings (DASIP), Cracow* (2015)
68. M. Skelin, E.R. Wognsen, M.C. Olesen, R. Hansen, K. Larsen, Model checking of finite-state machine-based scenario-aware dataflow using timed automata, in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)* (2015), pp. 1–10
69. M. Skelin, E.R. Wognsen, M.C. Olesen, R. Hansen, K. Larsen, Towards translating FSM-SADF to timed automata, in *1st International Workshop on Investigating Dataflow in Embedded Computing Architectures (IDEA), Amsterdam* (2015), pp. 13–16
70. M. Skelin, M. Geilen, F. Catthoor, S. Hendseth, Worst-case performance analysis of SDF-based parametrized dataflow. Microprocess. Microsyst. **52**, 439–460 (2017). Elsevier, invited paper, (online) Dec. 2016
71. M. Skelin, M. Geilen, F. Catthoor, S. Hendseth, Parameterized dataflow scenarios. IEEE Trans. Comput. Aided Des. **CAD-36**(4), 669–682 (2017)
72. N. Tack, G. Lafruit, F. Catthoor, R. Lauwereins, Platform independent optimisation of multi-resolution 3D content for enabling universal media access. Vis. Comput. **22**(8), 577–590 (2006)
73. M. Temmerman, E. Daylight, F. Catthoor, S. Demeyer, T. Dhaene, Optimizing data structures at the modeling level in embedded multimedia. J. Syst. Architect. **53**(8), 465–550 (2007)
74. B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, S. Stuijk, A scenario-aware data flow model for combined long-run average and worst-case performance analysis, in *Proceedings of the 4th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)* (IEEE Computer Society Press, Washington, 2006), pp. 185–194
75. F. Vandeputte, L. Eeckhout, K. De Bosschere, A detailed study on phase predictors, in *Proceedings of the 11th International Euro-Par Conference* (Springer, Berlin, 2005), pp. 571–581
76. F. Vandeputte, L. Eeckhout, K. De Bosschere, Exploiting program phase behavior for energy reduction on multi-configuration processors. J. Syst. Architect. **53**(8), 489–500 (2007)
77. P. van der Mark, L. Wolters, G. Cats, Using semi-Lagrangian formulations with automatic code generation for environmental modeling, in *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia* (2004), pp. 229–234
78. A. Vogel, B. Kerherve, G. von Bochmann, J. Gecsei, Distributed multimedia and QoS: a survey. IEEE Multimedia **2**(2), 10–19 (1995)
79. E. Wandeler, L. Thiele, Characterizing workload correlations in multi processor hard real-time systems, in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (IEEE Computer Society Press, Washington, 2005), pp. 46–55
80. I. Wegener, Integer-valued DDs, in *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications, chap. 9 (Society for Industrial and Applied Mathematics, Philadelphia, 2000)

81. P. Yang, Pareto-optimisation based run-time task scheduling for embedded systems. Doctoral Dissertation, ESAT/EE Department, K.U. Leuven (2004)
82. P. Yang, F. Catthoor, Pareto optimization based run-time task scheduling for embedded systems, in *Proceedings of Workshop on Hardware/Software Co-Design and International System-Level Synthesis Symposium (Codes-ISSS), San Diego* (2003), pp. 120–125
83. P. Yang, F. Catthoor, Dynamic mapping and ordering tasks of embedded real-time systems on multi-processor platforms, in *8th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. Lecture Notes in Computer Science (Springer, Berlin, 2004), pp. 167–181
84. P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, R. Lauwereins, Energy-aware runtime scheduling for embedded multi-processor SOCs. IEEE Des. Test Comput. **18**(5), 46–58 (2001). Special issue on "Application-specific multi-processor mapping"
85. P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, R. Lauwereins, Managing dynamic concurrent tasks in embedded real-time multimedia systems, invited paper in *Proceedings of 15th ACM/IEEE International Symposium on System-Level Synthesis (ISSS), Kyoto* (2002), pp. 112–119
86. Y. Yassin, P.G. Kjeldsberg, F. Catthoor, H264/AVC system scenario framework evaluation on EFM32, in *Proceedings 22th European Conference on Circuit Theory and Design, ECCTD, Trondheim* (2015)
87. Y. Yassin, P.G. Kjeldsberg, F. Catthoor, Dynamic hardware management of the H264/AVC encoder control structure using a framework for system scenarios, in *Proceedings of Euromicro Symposium on Digital System Design (DSD), Limassol* (2016), pp. 222–229
88. Y. Yassin, P.G. Kjeldsberg, F. Catthoor, Techniques for scenario prediction and switching in system scenario based designs, poster presentation at PhD Forum in *20th ACM/IEEE Design and Test in Europe Conference (DATE), Lausanne* (2017)
89. Y. Yassin, F. Catthoor, P.G. Kjeldsberg, A. Perkis, Techniques for dynamic hardware management of streaming media applications using a framework for system scenarios, in *Microprocessors and Microprogramming* (Elsevier, Amsterdam, 2018)
90. C. Ykman-Couvreur, E. Brockmey, V. Nollet, T. Marescaux, F. Catthoor, H. Corporaal, Design-time application exploration for MP-SoC customized run-time management, in *Proceedings of International System-on Chip Symposium (SoC), Tampere* (2005), pp. 66–73
91. C. Ykman-Couvreur, V. Nollet, T. Marescaux, E. Brockmey, F. Catthoor, H. Corporaal, Fast multi-dimension multi-choice knapsack heuristic for MP-SoC run-time management, in *Proceedings of International System-on Chip Symposium (SoC), Tampere* (2006), pp. 195–198
92. D. Yokota, S. Chiba, K. Itano, A new optimization technique for the inspector-executor method, in *Proceedings of International Conference on Parallel and Distributed Computing Systems (PDCS), Cambridge* (2002), pp. 706–711

Chapter 3
System-Scenario-based Design Techniques in the Presence of Data Variables

Elena Hammari, Yahya H. Yassin, Iason Filippopoulos, Francky Catthoor, and Per Gunnar Kjeldsberg

1 Introduction and Context

So far in this book, the system scenario design methodology has been outlined for use in cases where control variables decide the behavior and govern the changes between scenarios. For applications where data variables influence the behavior, these techniques are not fully sufficient. In this chapter, we will describe the necessary adaptation needed for the system scenario approach to work in the presence of data variables. Since the most important changes take place in the scenario identification phase at design-time, the main part of this chapter deals with this step of the methodology.

Figure 3.1 shows a simplified code snippet illustrating the difference between a control variable and data variable dependency. The code to the left illustrates a control variable dependency. The variable p in this case has a limited value range, and it will be relatively easy to monitor the application behavior if p is later used, for example, as a loop parameter.

The code snippet to the right of Fig. 3.1 shows a simple example of a data variable dependent situation, where the value range of p equals the value range of a. If a is an

E. Hammari · P. G. Kjeldsberg (✉)
Norwegian University of Science and Technology, NTNU, Trondheim, Norway
e-mail: pgk@ntnu.no

Y. H. Yassin · I. Filippopoulos
Norwegian University of Science and Technology, NTNU, Trondheim, Norway

KU Levuen, Leuven, Belgium

F. Catthoor
IMEC and KU Leuven, Leuven, Belgium
e-mail: catthoor@imec.be

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6_3

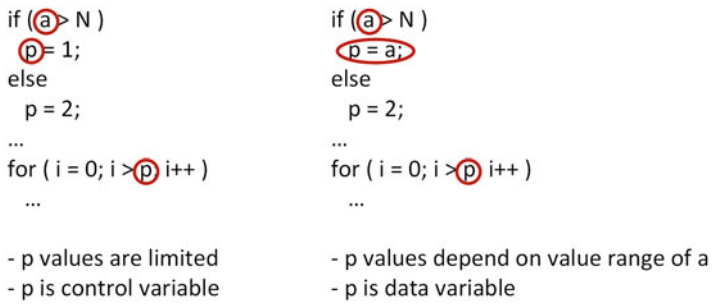


Fig. 3.1 Control vs data variable dependency

input parameter, and p is used as a loop parameter in succeeding computations, the loop size varies with the input parameter value range. Assuming the input parameter is an integer value with word length WL, then the value range of a will be equal to 2^{WL}, e.g., 4.3 billion for $WL = 32$. This value range would simply produce too much run-time overhead for the scenario prediction and switching mechanisms.

The rest of this chapter is organized as follows: Section 2 describes a general mathematical approach based on polyhedral modeling for the scenario identification process performed at design-time. This is followed by a section describing two more specific examples of scenario identification (Sect. 3). Having covered this design-time step in detail we turn to sections on techniques for run-time scenario detection (Sect. 4) and switching (Sect. 5). Most of the application examples used in these sections are relatively simple, in order to demonstrate the different concepts. In Sect. 6 we demonstrate the full effect of the methodology through a real-life application demonstrator. Finally, we present our conclusions related to data variable system scenarios in Sect. 7.

2 Scenario Identification Through Polyhedral Partitioning of the Parameter Space

The design-time scenario identification has to take into account the special requirements of data variable dependencies. In this section, we will show a mathematical approach for scenario identification based on clustering of run-time situations (RTSs) through polyhedral partitioning of the parameter space. We start with a definition of the scenario cost, followed by an algorithm for polyhedral scenario identification.

2.1 Scenario Cost Definition for Use in Polyhedral Partitioning

As presented in Chap. 2, an RTS is defined through the values of a set of RTS parameters. For each RTS i, its RTS *signature* is given by Eq. (3.1) below:

$$r(i) = \xi_1(i), \xi_2(i), \ldots, \xi_k(i); c(i) \tag{3.1}$$

containing parameter values $\xi_1(i), \xi_2(i), \ldots, \xi_k(i)$ and the corresponding task costs $c(i)$. That is, each run instance of each task will have its own RTS signature. The number, N, of RTS signatures will hence be very large.

Figure 3.2 now illustrates the theoretical concepts of our polyhedral scenario identification technique. Given k RTS parameters and N profiled RTS signatures from Eq. (3.1). If we assign one dimension to each RTS parameter, the resulting k-dimensional space will define all theoretically possible values for the RTS parameters in the application. We will call this space an *RTS parameter space*. When static max and min constraints on the values are added, the space reduces to one or several k-dimensional domain(s).

Assuming a $(k + 1)$-dimensional cost representation for each RTS, all signatures can then be plotted as points in a $k + 1$ dimensional space. In the profiling sequence, several identical signatures may exist, giving coinciding points in the space. The number of times a point repeats itself is useful information as it quantifies the probabilities of occurrence of each RTS.

With the representation above, the scenario identification task can be viewed as a distribution of points into S different groups, representing system scenarios, in such a way that the overall configuration cost is minimized. An RTS point i is assigned to scenario j whenever its cost $c(i)$ falls into that scenario's cost

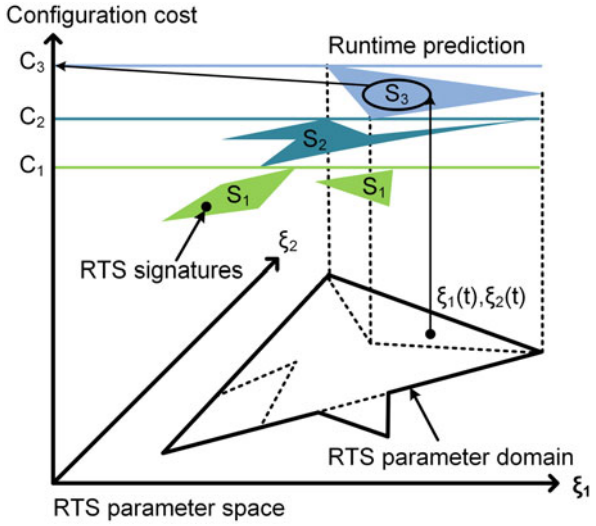


Fig. 3.2 RTS parameter space with additional configuration cost dimension

range $\{C(j)_{\min}, C(j)_{\max}\}$. The scenario cost ranges are determined by a balancing function that ensures that all scenarios have a sufficiently close probability to occur at run-time. In this way, rare system scenarios are avoided since their storage cost will exceed the gains of adding them. We measure this probability by the number of points, including the repeating ones, that each scenario contains and call it *scenario size*. Given a list r of RTS signatures sorted by descending cost, the scenario cost ranges are given by the indices corresponding to the integral number of the maximum scenario size N/S:

$$C(j)_{\max} = r\left((j-1)\cdot\frac{N}{S}+1\right) \tag{3.2}$$

$$C(j)_{\min} = r\left(j\cdot\frac{N}{S}\right) \tag{3.3}$$

The *cost* of scenario C_j is defined as the maximum cost of any of the RTS signatures that it includes: $C_j = C(j)_{\max}$.

The projection of scenarios onto the RTS parameter space (see Fig. 3.2) will produce $M \geq S$ regions that characterize the system scenarios in terms of RTS parameter values. Each region can be described as a polyhedron, and the run-time scenario prediction can be done by checking which polyhedron contains the RTS parameter values of the next RTS. Since we know which scenario the region belongs to, we can foresee that the next running cost will be no more than the cost of that scenario. Checking if a point lies inside a polyhedra is the classical data point location problem from the computational geometry domain [5]. The advantage of using it for prediction instead of a multi-valued decision diagram (MDD), typically used for control variable dependencies, is that it operates on/stores only the vertices of polyhedral regions, not the whole RTS parameter space.

This top-down approach can handle arbitrary large domains, provided that the number of regions stays reasonably low. Otherwise prediction overhead will grow. The number of regions depends on the number of system scenarios and the underlying structure of the system—the relationship between the cost locality of RTS points and the value locality of their RTS parameters.

As already mentioned in Chap. 2, the desired number of system scenarios is best defined by the user according to the characteristics of the application domain. Typically, this is limited to a few tens because beyond that the potential gains in better following the system dynamics are counterbalanced with the additional cost complexity of detecting and exploiting the (too) large set of possible system scenarios.

For many applications, a strong correlation is present in the locality of the RTS parameter values and the locality of the corresponding costs on the target DSP platform, resulting in a single region per scenario (see Fig. 3.4). The locality of parameters and the corresponding costs represents an important prerequisite for the efficiency of the current scenario identification technique. Moreover, we currently assume that there is a one-to-one correspondence between the cost of a scenario and

the system configuration. The scenario thus represents the system mapping. This is not a necessary assumption, however, and similar concepts of data variable scenarios are being explored in domains outside the system mapping context [4].

2.2 Algorithm for Polyhedral Scenario Identification

Our scenario identification algorithm, GENERATESCENARIOSET, is presented in Fig. 3.3. Line 2 is a preprocessing step, where profiled RTS signatures are sorted by their costs starting from the maximum cost. In Line 4 the worst-case system scenario is created. In Lines 6–8, the system scenario is filled with signatures having the next costs in the sorted sequence. When the size of the system scenario exceeds the maximum possible size, a new system scenario is created (Line 10).

The complexity of this algorithm can be calculated as given in Eq. (3.4), where N is the number of RTS signatures and S is the number of system scenarios. It depends on the complexity of the underlying geometric algorithms in the labeled functions. For the fixed number d of RTS parameters, the function NEWSCENARIO

```
GENERATESCENARIOSET(SET rtsSignatures, INT nrScenarios)
 1  solutions ← ∅
 2  SORTBYCOST(rtsSignatures)
 3  wcSignature ← rtsSignature(1)
 4  currentScenario ← NEWSCENARIO(wcSignature)
 5  MAXSCENARIOSIZE := rtsSignatures.size()/nrScenarios
 6  for signature in rtsSignatures do
 7     if (currentScenario.size < MAXSCENARIOSIZE) then
 8        currentScenario.ADDSIGNATURE(signature)
 9     else
10        systemScenario ← NEWSCENARIO(signature)
11        solutions.INSERT(systemScenario)
12     end if
13  end for
14  return solutions

NEWSCENARIO(RTSSIGNATURE signature)
   SCENARIO new
   new.size ← 1
   new.cost ← signature.cost
   new.paramRegion ← signature.paramValues
   return new

ADDSIGNATURE(RTSSIGNATURE signature)
   thisScenario.size ← thisScenario.size + 1
   thisScenario.paramRegion ←
                          CONVEXHULL(thisScenario.paramRegion, signature.paramValues)
   return
```

Fig. 3.3 Base system scenario identification algorithm

has a constant complexity O(d) as it only copies the value of each parameter in a single RTS signature to a scenario region.

$$\text{Complexity} = N \cdot \text{O}(\text{AddSignature}) + \sum_{i=1}^{S} \text{O}(\text{NewScenario}) \quad (3.4)$$

The function AddSignature performs a ConvexHull operation on the existing border of the scenario and the projection point of the new RTS signature onto the RTS parameter space. For a 2 or 3-dimensional RTS parameter space an incremental convex hull algorithm has the complexity O($n \log n$) [11], where n is the final number of processed points, which here equals to the scenario size, N/S in the worst case. The convex hull of a polygon has the expected number of $v = \text{O}(\log n)$ vertices and many of them may lie very close to each other. To limit the number of vertices in the hull for faster run-time prediction, we modify the algorithm, such that it calculates the distance between the points on the hull and removes those that are closer than $L/v_{\max}$, where L is the perimeter of the hull, and $v_{\max}$ is a user defined constraint of the maximum number of vertices in the prediction polyhedra. For the application that we investigate a reasonable value of this parameter could be 10 (see Fig. 3.4).

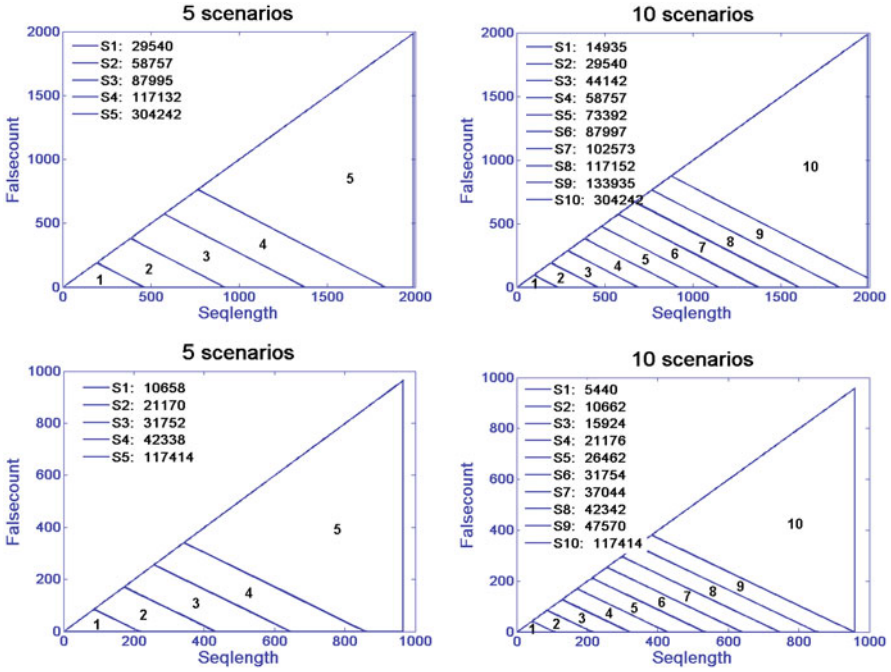


Fig. 3.4 System scenarios for the application version I (top) and II (bottom). Numbers in the top left corners are the execution times of scenarios in clock cycles

The total complexity of the algorithm is $O(N \log \frac{N}{S}) + O(S \cdot d) = O(N \log N)$, considering that $S, d \ll N$. Recall that this scenario identification algorithm is run only in the design phase of the embedded system. The run-time prediction of the next scenario is equivalent to a point location problem in the polyhedral partitioning of the parameter space. The time complexity of the point location problem is $O(\log v)$, where $v = S \cdot v_{\max}$ is the total number of vertices in the partitioning, given $O(v \log v)$ memory space. To compare, an MDD for d parameters with l distinct values has l^d states and a query time of $O(d \cdot l)$, where $l \gg v$.

For $d > 3$, i.e., for systems with more than 3 parameters, the complexity of convex hull and Boolean set operations on polytopes increases exponentially in d [2], similar to MDD. It remains an open research area to find an efficient representation of polytopes in higher dimensions that will decrease the complexity of the algorithms operating on them. The complexity does not increase exponentially when the number of possible parameter values increases, however, the way it does for MDD.

The function AddSignature in Fig. 3.3 produces convex scenario projections in the RTS parameter space. However, in some situations concave scenario projections are preferable, e.g., when the inherent correlation between the RTS parameter values and the corresponding costs has a concave shape. However, large overhead may incur since algorithms processing concave polyhedra are much more complex. A possible solution is to split the concave projection into a set of convex polyhedra at design-time and apply convex hull algorithms. The separate polyhedra still require additional storage and processing time that should be kept low. To achieve that, a restriction must be made on the number of reflex angles in the concave projection, and also a careful consideration of the cost trade-offs must be made. The refinement step is performed after a system scenario has been completed with signatures. At this stage, the distribution of the points and the final convex hull are available and are used in the refinement step. The other considerations are the goals of our future work.

For system scenario prediction it is crucial that the identified system scenarios are disjoint in the RTS parameter space. To satisfy this requirement, the designer must select RTS parameters carefully and ensure that each scenario border tightly follows the underlying distribution of the points through the refinement step. If (small) overlaps still exist in the resulting system scenario set, they can be eliminated by moving all signatures in the overlapping region to the higher cost system scenario. This approach ensures that the predicted system scenario cost is never underestimated. Overlap removal is described in detail in our work [7].

2.3 Experimental Evaluation of Algorithm for Polyhedral Scenario Identification

We have evaluated our scenario identification algorithm on several versions of a Lyapunov exponent calculator used in an epileptic seizure predictor [9, 10]. We now present results for execution time optimization using system scenarios. The improved execution time can be exploited for reconfiguration in several ways. DVFS can be applied, possibly in combination with rescheduling to allow other tasks to run in the idle time. On run-time reconfigurable multi-processor platforms, remapping of tasks is possible to achieve an overall optimized execution.

We have run tests on three different setups, displayed in Table 3.1. Throughout the tests we have varied: (a) the version of the application, (b) the platform on which the execution time was measured, and (c) the input database for application profiling. This results in different characteristics, which represent distinct benchmarks to test our algorithm performance. Figures 3.4 and 3.5 show the identified system scenarios. The experiments in Fig. 3.4 are performed on the embedded platform, CoolBio DSP [1], and two different application versions are used. In both versions, the same two variables have the greatest impact on the application execution time—*Falsecount* and *Seqlength*. They are therefore selected as RTS parameters giving a two-dimensional RTS parameter space. The figures show that there is a clear correlation between these parameters and the execution workload (in clock cycles)—giving non-overlapping projections for each system scenario onto the RTS parameter plane. The two application versions result in different sizes of the RTS parameter domain for Experiment 1 and Experiment 2. The identified system scenarios appear to be very similar, however, approximately a scaled version of each other. As already indicated earlier, a number of 5–20 system scenarios are preferred by users in scenario-based systems. In Fig. 3.4 we only show results for 5 and 10 scenarios. The trend is identical for 15 and 20.

Experiment 3 demonstrates the scenario refinement step. To get concave distributions of points in the scenario projections, we rerun scenario identification in application version I on a general purpose desktop. Figure 3.5 presents the 5 identified system scenarios. The subfigures show each scenario projection separately along with its respective RTS point distribution. Notice that the values on the axis

Table 3.1 Experimental setups

Experiment	Application version	Platform	Database
1	I, settings: nsize = 2048, dimm = 7, tau = 4, evolv = 12, idist = 20	CoolBio DSP	6 h continuous EEG w/seizures
2	II, settings: nsize = 1000, dimm = 4, tau = 4, evolv = 6, idist = 12	CoolBio DSP	6 h continuous EEG w/seizures
3	I, settings: nsize = 2048, dimm = 7, tau = 4, evolv = 12, idist = 20	General purpose	200 EEG samples from epileptogenic zone, no seizures

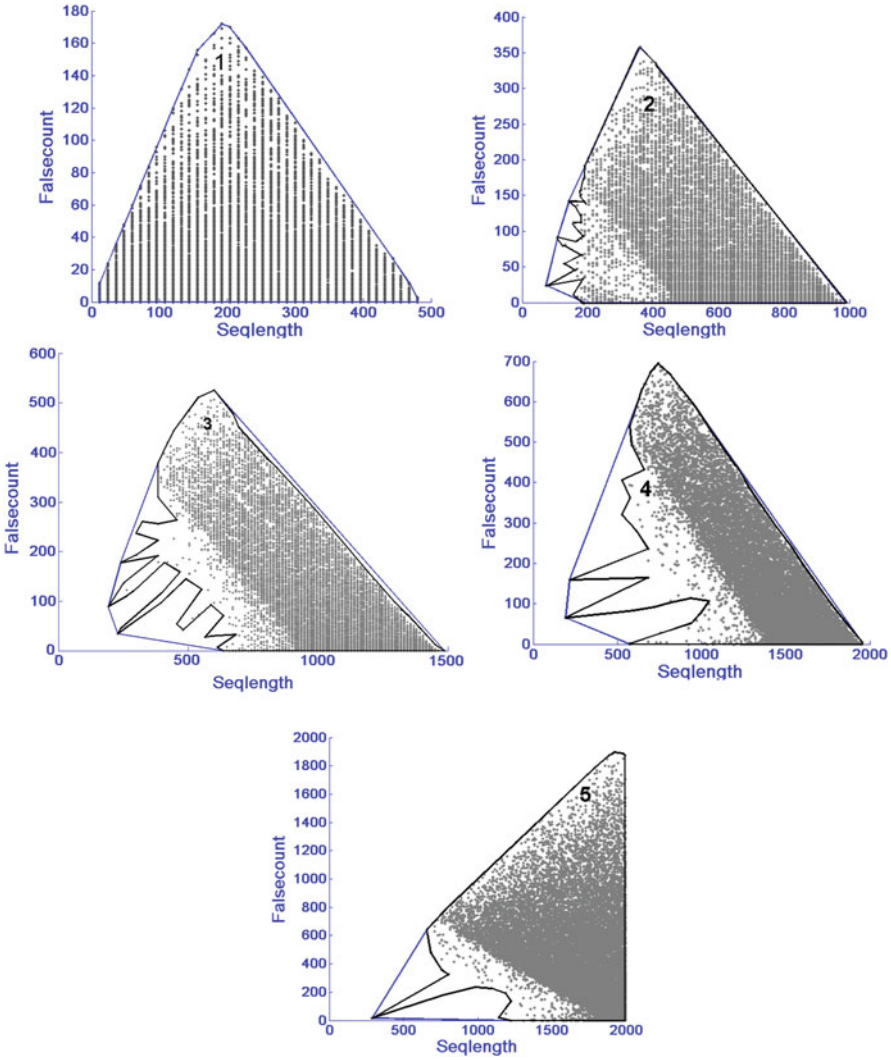


Fig. 3.5 Experiment 3. Convex scenario projections and their concave refinement. Numbers in the top left corner are the execution times of scenario projections in μs

are different in the subfigures. Concave refinement is performed on scenarios 2–5. Although a much smaller database is used for this experiment, the real scenario borders from Fig. 3.4 can be discerned in the point distributions of Fig. 3.5. The distortion in the scenario borders is caused by the noise from the nondeterministic platform. While the convex hull is strongly affected by this noise, the concave hull comes closer to the real scenario borders and can potentially improve the execution time of this system-scenario-based design.

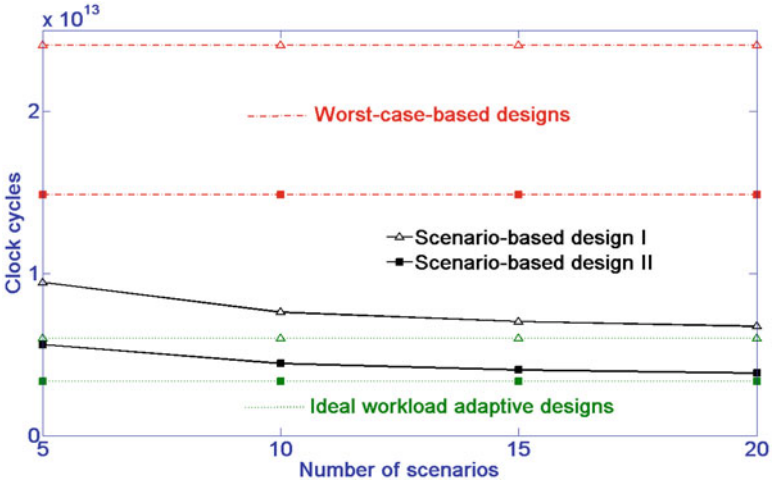


Fig. 3.6 Total execution time of the largest Lyapunov exponent calculation (versions I and II) with different number of system scenarios in the 6 h continuous EEG recording

Figure 3.6 compares execution workload for running the system with and without system scenarios on CoolBio embedded platform. It also shows the execution workload of the theoretically optimal workload-adaptive design, which is not realizable in practice. The results are presented for both Experiments 1 and 2, i.e., application versions I and II, respectively. Between 61% and 72% gain can be achieved with 5–20 system scenarios. With 5 system scenarios the total execution workload of the systems is still situated well above the theoretically best solution: 58% for system I and 67% for system II. When the number of scenarios is increased, the total execution time of both systems reduces towards the theoretical limit and becomes at 20 scenarios less than 13% and 15% above the theoretically best solution for version I and version II, respectively. This is promising, and shows that near-optimal solutions can be achieved with a limited number of scenarios.

The results presented here demonstrate the practical feasibility of the proposed technique and show that it is possible to reach near-optimal execution time with a limited number of scenarios. Future work includes optimization of scenario borders to realize a trade-off between overestimation and run-time prediction/switching complexity.

3 Scenario Identification Based on Specific Cost Parameters

In this subsection we will show examples of design-time scenario identification based on two other important system cost parameters: memory requirement on the hardware platform and image frame sizes in the application. The image size can

also be generalized to sample resolution, and hence to quality of service of the application.

3.1 *RTS Clustering Based on Memory Size and Frequency of Occurrence*

The previous section gave an example of scenario identification based on using execution time as the cost function. We will now show how another platform characteristic can be used as cost function for data variable system scenarios, in this case memory size. Similarly to execution time, memory size can also be related to energy consumption, if unused memory blocks can be turned off dynamically at run-time.

Figure 3.7 shows a motivational example with two image related multimedia benchmarks and an input database consisting of a variety of images. The memory requirements in each case are driven by the current input image size, which is classified as a data variable due to the wide range of its possible values. Depending on the application, the whole image or a region of interest is processed. The number of possible sizes is too large to embed in a control variable, and must hence be handled as a data variable scenario identification case. Other applications have other input data variables deciding the memory requirement dynamism, e.g., the channel signal-to-noise ratio in the case of an encoding/decoding application.

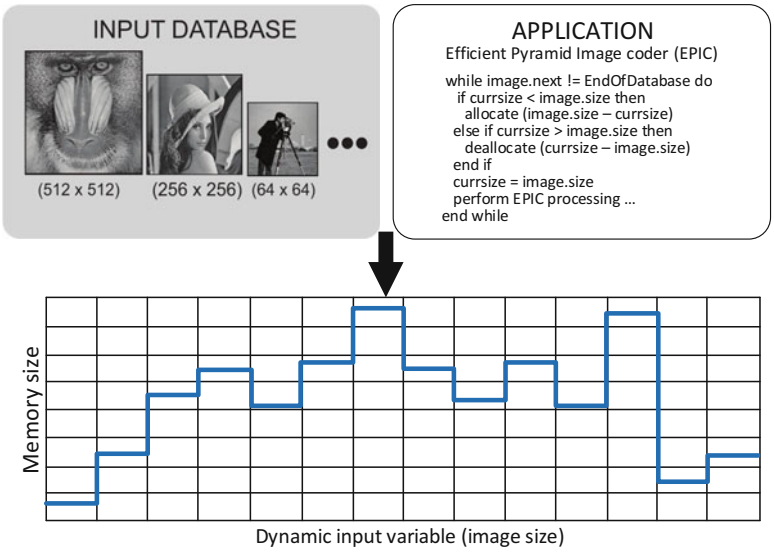


Fig. 3.7 Profiling results based on application code and input data

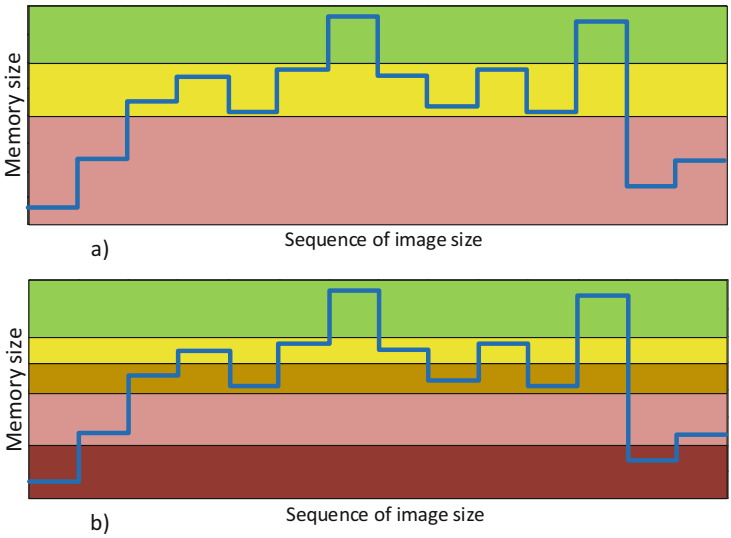


Fig. 3.8 Clustering of profiling results into three (**a**) or five (**b**) system scenarios

The system scenario identification now consists of a clustering of the profiled memory sizes into groups with similar characteristics. Clustering is necessary, because it will be extremely costly to have a different scenario for every possible size, due to the number of memories needed. We have not applied the formal polyhedral framework and the supporting tools of Sect. 2. This will be part of future work. Instead we manually apply a similar approach to illustrate the concepts.

Clustering neighboring RTSs is a rational choice, since two instances with similar memory need to have similar energy consumption. In Fig. 3.8 the clustering is based both on their distance on the memory size axis and the frequency of their occurrence. Consequently, the memory size is split unevenly with more frequent RTSs having a shorter memory size range. In the case of a clustering to three system scenarios the space is divided in the three differently colored areas depicted in Fig. 3.8a. Due to the higher frequency of RTSs in the yellow area that system scenario has a shorter range compared with its neighboring scenarios. Such clustering is better than an even splitting because the energy cost of each system scenario is defined by the upper size limit, as each scenario should support all RTSs within its range. Consequently, the overhead for the RTSs in the yellow area is lower compared to the overhead in the two other areas.

The same principle applies also when the number of system scenarios is increased to five, as depicted in Fig. 3.8b. The frequency sensitive clustering results in two short system scenarios that contain four RTSs each and three wider system scenarios with fewer RTSs. The number of system scenarios should be limited mainly due to two factors. First, implementation of a high number of system scenarios in a memory platform is more difficult and complex. Second,

the switching between the different scenarios involves an energy penalty that could become significant when the switching takes place frequently.

The memory size and the frequency of each RTS are not the only two parameters that should be taken into consideration during the system scenario identification. The memory size of each RTS results in a different energy cost depending on the way it is mapped into memory. The impact of the different assignment possibilities is included into the clustering by introduction of energy as a cost metric. The energy cost for each RTS is calculated using a reference platform with one to N memory banks. Increasing the number of memory banks results in lower energy per access since the most accessed elements can be assigned to smaller and more energy-efficient banks. Unused banks can be switched off.

In the system scenario methodology, Pareto curves are used to capture alternative system configurations within a scenario [13]. In our work, a Pareto space is used for clustering that also includes the energy cost metric. For each RTS all different assignment options on alternative platform configurations are studied. Memory platform knobs are different sets of memory banks that are turned on and off. A Pareto curve is constructed for each RTS that contains the optimal assignment for each platform configuration. Hence, suboptimal assignments and assignments that result in conflicts are not included in the Pareto curve. In Fig. 3.9 four Pareto curves, each corresponding to a different RTS, are shown together with energy cost levels corresponding to different platform configuration and data-to-memory assignment decisions. Three non-optimal mappings are also shown in Fig. 3.9 for illustration. They are not part of the Pareto curve and consequently not included in the generation of scenarios. Pareto curves are clustered into three different system scenarios based again both on their memory size differences and frequency of occurrence. Clustering of RTSs using Pareto curves is more accurate compared to the clustering depicted in Fig. 3.8, as it includes data-to-memory assignment options in the exploration.

The system scenario identification step includes the selection of the data variables that determine the active system scenario. This can be achieved by careful study of the application code, combined with the application's data input. The variable selection is done before clustering of RTSs into scenarios. For the choice of identification variables, there is a trade-off between the complexity and the accuracy of the scenario detection step. On one hand, if the identification is done using a group of complex variables and their correlation, there is a number of calculations needed in order to predict the active scenario. On the other hand, if the value of a single variable is monitored for scenario identification, the scenario detection is straightforward. In our case the gray-box model reveals only the code parts that will influence memory usage, so that data variables deciding memory space changes can be identified. An example of this is a non-static variable that influences the number of iterations for a loop that performs one memory allocation at each iteration. In the depicted example the system scenario detection data variable is the input image height and width values.

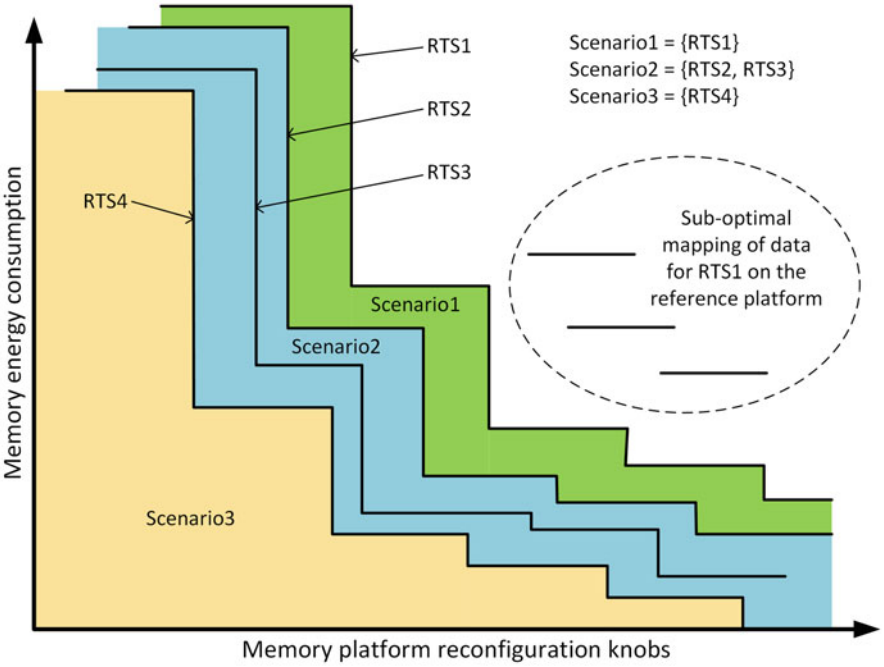


Fig. 3.9 Clustering of Pareto curves

3.2 Clustering of RTSs Based on Image Size and Set of Available Platform Configuration Knobs

The final illustration of the power of data variable based scenario identification uses streaming media applications. The behavior is again profiled with different input data in order to detect different RTSs. Our scenario identification approach takes into account available and relevant application domain parameters. This is typically the stream of input data, the application code handling this data, long input sequence which we profile in order to identify the dynamism in the workload, application run-time due to while loops/conditions, data variable dependencies, and storage requirements. Again, the number of options in the parameter settings is typically too high to be incorporated effectively in a control variable approach.

We now use the large real-life H.264/AVC video encoder to demonstrate the methodology. Its source code was obtained from the MediaBench II website [12]. It will be presented in more detail in Sect. 6. This model is profiled at design-time in order to explore the dynamic range of loop iteration count variations for various input frame sizes. The H.264/AVC encoder control structure is extracted from the source code, and modeled under the assumption that all memory accesses take one clock cycle. Listing 3.1 shows an example of a nested loop structure found in the H.264/AVC encoder.

```
// Nested loops in H264/AVC encoder control structure
...
for (j = -img_pad_size; j < size_y + img_pad_size; j++)
   ...
   for (i = -img_pad_size; i < size_x + img_pad_size; i++)
      ...
for (i = 0; i < (size_x + 2 * img_pad_size) * 2; i++)
   ...
   for (j = 2; j < maxy - 2; j++)
      ...
   for (j = maxy - 2; j < ypadded_size; j++)
      ...
for (j = 0; j < je2 + 2; j += 2)
   for (i = 0; i < ie2 - 2; i += 2)
      ...
   for (i = ie2 - 2; i < ie2 + 1; i += 2)
      ...
for (i = 0; i < ie2 + 2; i++)
   ...
   for (j = 0; j < je2 - 2; j += 2)
      ...
   for (j = je2 - 2; j < je2 + 1; j += 2)
      ...
for (j = 0; j < size_y/4; j++)
   ...
   for (i = 0; i < size_x/4; i++)
      ...
...
```

Listing 3.1 Part of control structure for H.264/AVC encoder

The H.264/AVC encoder is profiled for different voltage and frequency settings in order to find optimal DVFS configurations in combination with race-to-halt (RTH). A SAM4L board from Atmel [3] is used for the profiling, being a representative example of a mobile platform containing an energy friendly CPU with limited on-chip SRAM. One advantage of the SAM4L-board is the application control of active voltage states, making it possible to dynamically tune the hardware for the running application. Our profiled results show that the optimal energy consumption is achieved if the maximum available frequency setting for each voltage level is selected. This results in two scenarios, PS0 (1.8 V and 40 MHz) and PS1 (1.2 V and 12 MHz). Further details on the experimental setup can be found in Sect. 6.

Table 3.2 shows an evenly distributed selection of true 16:9 resolutions, which our SAM4L board can process within a one second time frame using our extracted model. It also shows the corresponding scenario selections and calculated run-times, based on our available power modes on the SAM4L board. In Table 3.2, the Scenario* column represents the scenario distribution for a platform with more available voltage settings. This will typically give more potential for optimal adaption, and thus further reductions in energy consumption. Details are, however, outside the scope of this book. See [17] for additional details.

The largest frame size the platform can handle from our selection is measured to be 816×459. The required run-time (in clock cycles) for each frame size is estimated by counting all loop iterations and multiplying with all instructions within each loop based on the program disassembly of the extracted model. These numbers are compared to the measured run-time for each frame size. From the

Table 3.2 Scenario assignment based on run-time

Input frame size	Run-time (million cycles)	Scenario	Scenario*
816 × 459	34.70	0	0
752 × 423	29.54	0	0
688 × 387	24.79	0	1
624 × 351	20.46	0	2
560 × 315	16.54	0	2
496 × 279	13.04	0	3
432 × 243	9.96	1	4
368 × 207	7.29	1	4
288 × 162	4.54	1	5
224 × 126	2.80	1	5

measured results, we observed that the correct cycle count could be approximated by multiplying the number of all loop iterations for any input frame size with a factor of 4.22. This factor corresponds to the additional assembly instructions needed to determine the loop length and initiate for-loops before they are executed. The number of all loop iterations is calculated as shown in Lines 2–7 in Listing 3.2. The total number of iterations are combined together with other parameters for all the nested loops in the application, resulting in the loopcnt variable. This loopcnt variable is multiplied by the approximated factor of 4.22. The separation of the approximated factor allows for easy portability of the scenario mechanism to other platforms, because other platforms might implement the loop handling instructions differently. This difference would result in a higher or lower factor. The largest frame processing time is therefore approximated to 34.7 million clock cycles in an un-optimized setting where no pipelines are utilized. We use this frame size processing time as our worst-case scenario, and we exploit it in combination with DVFS and RTH for lower frame sizes. We benefit from DVFS if the processing time of a smaller frame size with a lower frequency setting is less than the processing time of our worst-case frame size. Therefore, we have to take our available system knobs into consideration when we classify our system scenarios.

Our scenario selection will be dependent on the platform's running frequency. In PS1 the platform is processing a frame 70% slower than in PS0. Therefore, frame sizes that need more than 30% of the run-time of our worst-case frame size are processed with PS0 (Scenario 0), and smaller frame sizes are processed with the PS1 configuration (Scenario 1).

Our arguments above are generalized in Eq. (3.5), where we set the Scenario 1 upper limit for the maximum frame processing time in PS1 to be equal to the frame processing time of our largest frame (816 × 459) in PS0. The Scenario 1 upper limit is calculated as shown in Eq. (3.6). In Eqs. (3.5) and (3.6), $T_{WCframe}$ is the execution time in seconds of the worst-case frame size. T_{sc_1} is the maximum allowed execution time for Scenario 1 in seconds. LC_{sc_0} and LC_{sc_1} are the total number of loop iterations for a specific frame in PS0 or PS1, respectively. β is the approximated scale factor used to get the total number of clock cycles, and f_{sc_0} is the platform's

```
// Monitoring of RTS parameters
long loopcnt1 = (size_y + img_pad_size + img_pad_size) * (2 + (3*(size_x +
    img_pad_size + img_pad_size)));
long loopcnt2 = ((size_x + 2 * img_pad_size) * 2) * (5 + ((3*(maxy - 2 - 2)) +
    (3*(ypadded_size - maxy + 2))));
long loopcnt3 = ((je2 + 2)/2) * (2 + (((ie2 - 2)/2) + 2));
long loopcnt4 = (ie2 + 2) * (((je2 - 2)/2) + 2);
long loopcnt5 = (size_y/4) * (1 + (6*(size_x/4)));
long loopcnt = loopcnt1 + loopcnt2 + loopcnt3 + loopcnt4 + loopcnt5;

// Update combined RTS parameter
(g_rts_data)[0] = loopcnt;

// Scenario detection/prediction
AMU_1(scenario_old, scenario, rts_data[], g_pareto_lc[]);

// Nested loops as shown in previous listing
...
```

Listing 3.2 Functions combining data- dependent RTS parameters

main clock frequency when Scenario 0 is active. In other words, if the required frame processing time is above 30% of our worst-case frame processing time, then the platform switches from Scenario 1 to 0. Based on Eq. (3.5), we find the Scenario 1 limit in Eq. (3.6) without involving the β scale factor. This simplification makes our method portable to other platforms with a different β scale factor. Equation (3.6) shows the limit 2,466,807, which corresponds to 10.4 million clock cycles of actual run-time (after multiplying with the 4.22 approximated scale factor β). That is, all frame sizes with a run-time above 10.4 million clock cycles are assigned to Scenario 0 in Table 3.2.

$$T_{WCframe} = T_{sc_1} = \frac{\beta \cdot LC_{sc_0}}{f_{sc_0}} = \frac{\beta \cdot LC_{sc_1}}{0.3 \cdot f_{sc_0}} \tag{3.5}$$

$$LC_{sc_1} = 0.3 \cdot LC_{sc_0} = 0.3 \cdot 8,222,691 = 2,466,807 \tag{3.6}$$

4 Scenario Detection

While scenario identification is a pure design-time step, scenario detection is split between design-time and run-time. In this section we will show how data variables influence both how the scenario detection mechanism is developed at design-time and how it is used at run-time. According to the overall systems scenario design flow, the boundaries of the scenario set have already been defined, in this case using the data variable identification step presented in Sects. 2 and 3. Still we need to provide a system framework for the actual run-time detection. This will be presented in the following.

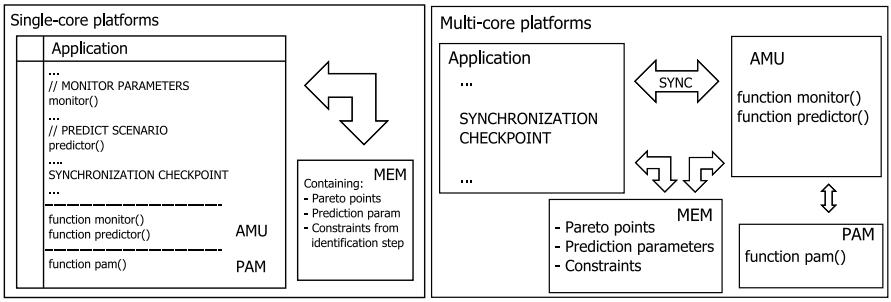


Fig. 3.10 Proposed FSS framework with single-core solution to the left and multi-core to the right

4.1 Scenario Prediction Using Application Monitoring Unit

Figure 3.10 illustrates a generic framework for system scenarios (FSS) that enables use of the system scenarios approach through an application monitoring unit (AMU) and a platform adaption manager (PAM) for single-core and multi-core architectures. The AMU part of this framework for system scenarios (FSS) will be presented here, while the PAM will be presented in Sect. 5.1.

In single-core architectures, the AMU is implemented in software as lightweight inline functions with minimal overhead. In homogeneous or heterogeneous multi-core architectures, it could be implemented in hardware running in parallel with the application, or as co-processors where parts of the AMU are implemented in software. This makes the framework also applicable for customized single-core architectures with the AMU implemented as customized units. The synchronization checkpoints in Fig. 3.10 are locations in the application code where the RTS parameters monitored by the prediction mechanisms are evaluated. Based on the outcome of the evaluation, the switching mechanism triggers a platform reconfiguration. Where the synchronization checkpoint is placed in the code depends on the behavior of the application, and a trade-off must be made based on how often we wish to reconfigure the platform versus the overhead cost of the monitoring and reconfiguration mechanisms.

4.2 Scenario Prediction Through Precomputation

The AMU introduced in the previous subsection can use different techniques to monitor RTS parameters and predict the upcoming scenario. The easiest way is to directly use the equations that define the data variable scenario boundaries, as derived in Sect. 2. However, in some applications the parameters present in these equations are not yet known at the moment when the potentially active next scenario needs to be detected. The parameters then need to be predicted upfront,

```
1  // Application Monitoring Unit (AMU) called in previous listing
2  inline void AMU_1(scenario_old, scenario, rts_data[], g_pareto_lc[]) {
3     if (rts_data[0] > g_pareto_lc[0])
4        scenario = DVFS_40MHz_1_8V;
5     ...
6     else if (rts_data[0] > g_pareto_lc[2])
7        scenario = DVFS_12MHz_1_2V;
8
9     if (scenario_old != scenario ) //  Only if change in scenario
10       DVFS_SAM4L(scenario);
11 }
12
13 // Platform Adaption Unit (PAM) called from AMU above
14 inline void DVFS_SAM4L(scenario){
15    switch (scenario)
16       case DVFS_40MHz_1_8V {
17          set_scenario_voltage(PS_0);
18          set_scenario_clk(CLK_40MHZ);
19       }
20       case DVFS_12MHz_1_2V {
21          set_scenario_clk(CLK_12MHZ);
22          set_scenario_voltage(PS_1);
23       }
24 }
```

Listing 3.3 Code for application monitoring unit and platform adaption manager for H.264/AVC encoder on the SAM4L-board

requiring some form of precomputation. In this subsection we will illustrate this precomputation approach for a realistic application context. Additional details of this technique can be found in [8].

The input data variable dependency of the H.264/AVC introduced in Sect. 3.2 is shown in Listing 3.2, where inline application dependent equations are pre-calculated (Lines 2–7), before a global lookup table is updated with the combined RTS parameter (Line 10) as part of the overall AMU. This RTS parameter is then read by the AMU_1 function, which detects the next scenario (Line 13). The application dependent calculations in Lines 2–6 correspond to succeeding (nested) for- and/or while-loops in the encoder control structure, which are combined together in Line 7 to estimate the required frame processing time in terms of loop iterations.

In our particular case it would have been enough to compare the loopcnt variable with the threshold between Scenario 0 and 1 directly, and use the result to determine the next scenario. However, we have generalized this comparison by using the AMU_1 function shown in Listing 3.3. This lightweight inline function compares the global RTS parameter lookup table values (rts_data[]) with global scenario lookup table values (g_pareto_lc[]) in order to determine which scenario the platform will adapt to. This generalization takes into account multiple scenario parameters and RTS parameters. The use of lookup tables simplifies an extension to multiple independent RTS parameters and/or scenarios. With the scenarios in column Scenario* of Table 3.2, we would get five cells in the g_pareto_lc[] lookup table. The values in the global scenario lookup table are found from our identification step through Eq. (3.6). If the value of the global RTS lookup table is greater than the value in the global scenario lookup table, then the platform changes its configurations to PS0 if it is in PS1, and vice versa. The PAM function is then

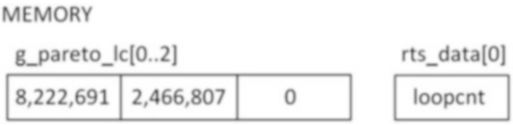


Fig. 3.11 Lookup table memory content used by AMU

called by the AMU and performs the platform reconfiguration using the DVFS system knob as described in Sect. 5. Figure 3.11 shows the memory content for the lookup tables used in Listing 3.3. rts_data contains the loopcnt value as calculated in Listing 3.2, e.g., 3,919,775 if the input frame size is 560×315. This corresponds to the values in Table 3.2 after multiplying the values with the β-factor.

5 Scenario Switching

The final run-time step, unless calibration is performed, is the actual switching of the platform configuration. This subsection presents the PAM as well as an example of the gain achieved performing the switching. It is here assumed that the upcoming scenario is detected as described in Sect. 4 and that it is different from the current scenario.

5.1 Scenario Switching Using Platform Adaptation Manager

Similar to the AMU described in Sect. 4.1 the PAM can be implemented either as lightweight inline functions in software for single-core solutions or as parallel hardware or in a co-processor for a multi-core solution. Figure 3.10 and Listing 3.3 give examples of both. As can be seen, the PAM is called from the AMU.

The PAM typically contains non-portable platform dependent functions, e.g., write to clock division registers, voltage regulation, and sleep mode activation. This is the main reason why the PAM is kept separate from the AMU, which can be made independent of the platform. In Listing 3.3 the switch case statements select between two voltages and frequencies. With the scenarios in column Scenario* of Table 3.2, the PAM would select between five voltages and frequencies.

5.2 Switching Gain Evaluation

If the platform switching has a substantial overhead, e.g., caused by slow voltage regulators or the delay of switching the memories on and off, the PAM may also include switching gain evaluation. In more detail, the switching mechanism implemented by the run-time manager includes the following actions:

Table 3.3 Relative dynamic energy for a range of memories with varying capacity and type

Type	Lines × wordlength	Dynamic energy [J]		Switching to active from	
		Read	Write	Deep [μJ]	Light [μJ]
MM	32 × 8	4.18×10^{-8}	3.24×10^{-8}	0.223	0.031
MM	32 × 16	6.79×10^{-8}	5.89×10^{-8}	0.223	0.031
MM	32 × 128	4.33×10^{-7}	4.31×10^{-7}	1.42	0.168
MM	256 × 128	4.48×10^{-7}	4.60×10^{-7}	1.70	0.171
MM	1024 × 128	5.11×10^{-7}	5.75×10^{-7}	2.81	0.179
MM	4096 × 128	9.60×10^{-7}	4.57×10^{-7}	9.01	0.457
SCMEM	128 × 128	2.5×10^{-7}	0.8×10^{-8}	1.51	0.045
SCMEM	1024 × 8	1.7×10^{-8}	0.6×10^{-8}	0.325	0.021

1. Determining the set of input data for which the upcoming scenario will be run.
2. Calculation of the energy consumption by processing the next set of input data on the currently active scenario (E1).
3. Calculation of the energy consumption by processing the next set of input data on its most energy-efficient scenario (E2).
4. Calculation of the energy penalty for switching to the configuration of the most efficient scenario (E3).
5. Evaluation of the expression: $E1 > E2 + E3$. If the energy cost of the current configuration (E1) is greater than the combined cost of the new configuration and the required switching (E2 + E3), then the decision is to switch. Otherwise, the switching decision is negative and the system stays on the currently running configuration.
6. Switching of the platform to its new configuration.

The PAM will in this case again mainly consist of comparisons of LUT values, since the evaluation of cost and gain can be performed as a design-time. Tables 3.3 and 3.4 give examples of relative energy costs of accessing different memories and of switching between sleep states [6].

The set of input data can typically be determined directly from the scenario that has already been detected/predicted according to the methodology presented in Sect. 4. At design-time, it is then only necessary to evaluate the energy costs of an average RTS belonging to a scenario, on the optimal (E2) and all other (E1) scenario configurations. This is doable since the number of scenarios is limited to a few tens. If the RTSs within a scenario differ in energy consumption, for example, because they have different run-time, the energy value found in the LUT for each scenario can be adapted according to the RTS signature encountered. This will require additional profiling at design time as well as evaluation of linear equations at run-time. Details of this are part of future work.

Together with design-time profiling information, the values in Tables 3.3 and 3.4 can be used to calculate E1, E2, and E3 above. All the important transactions on the platform that contribute to the overall energy are included in order to achieve as accurate results as possible. In particular:

Table 3.4 Relative static power for a range of memories with varying capacity and type

Type	Lines ×wordlength	Static leakage power per mode [W]			
		Active	Light sleep	Deep sleep	Shut down
MM	32 × 8	0.132	0.125	0.063	0.0016
MM	32 × 16	0.134	0.127	0.064	0.0022
MM	32 × 128	0.171	0.160	0.083	0.0112
MM	256 × 128	0.207	0.184	0.104	0.0293
MM	1024 × 128	0.349	0.283	0.189	0.102
MM	4096 × 128	0.95	0.708	0.544	0.396
SCMEM	128 × 128	0.083	0.057	0.027	0.0022
SCMEM	1024 × 8	0.042	0.028	0.014	0.0011

- N_{rd} is the number of read accesses
- E_{Read} is the energy per read
- N_{wr} is the number of write accesses
- E_{Write} is the energy per write
- T is the execution time of the application
- $T_{LightSleep}$, $T_{DeepSleep}$, and $T_{ShutDown}$ are the times spent in light sleep, deep sleep, and shut down states, respectively
- $P_{leak_{Active}}$ is the leakage power in active mode
- $P_{leak_{LightSleep}}$, $P_{leak_{DeepSleep}}$, and $P_{leak_{Shutdown}}$ are the leakage power values in light sleep, deep sleep, and shut down modes with different values corresponding to each mode
- $N_{SWLight}$, N_{SWDeep}, and $N_{SWShutDown}$ are the number of transitions from each retention state to active state
- $E_{LightSleep\ to\ Active}$, $E_{DeepSleep\ to\ Active}$, and $E_{ShutDown\ to\ Active}$ are the energy penalties for each transition, respectively.

$$
\begin{aligned}
E \;=\; \sum_{memories}^{all} (&N_{rd} \times E_{Read} \\
&+ N_{wr} \times E_{Write} \\
&+ (T - T_{LightSleep} - T_{DeepSleep} - T_{ShutDown}) \times P_{leak_{Active}} \\
&+ T_{LightSleep} \times P_{leak_{LightSleep}} \\
&+ T_{DeepSleep} \times P_{leak_{DeepSleep}} \\
&+ T_{ShutDown} \times P_{leak_{ShutDown}} \\
&+ N_{SWLight} \times E_{LightSleep\ to\ Active} \\
&+ N_{SWDeep} \times E_{DeepSleep\ to\ Active} \\
&+ N_{SWShutDown} \times E_{ShutDown\ to\ Active})
\end{aligned}
\tag{3.7}
$$

With the energy values calculated at design-time stored in a LUT, the PAM only needs to evaluate E1 > E2 + E3 to decide whether to switch or not.

6 Large-Scale Application Demonstrator

In Sect. 3.2 we briefly introduced the SAM4L board and the H.264/AVC video encoder. We will now describe the experimental setup in more detail and compare energy reduction results compared to a RTH approach without system scenarios.

6.1 Application, Platform, and Scenario System Settings

The SAM4L board from Atmel [3] is a representative example of a mobile platform because it contains an energy friendly CPU with limited on-chip SRAM. One advantage of the SAM4L-board is the application control of active voltage states, making it possible to dynamically tune the hardware for the running application.

During our experiments we measure the live current consumption and core voltage on the SAM4L board using a NI myDAQ measurement device [14]. The processor core voltage is measured directly on the vcc_mcu_core pin of the SAM4L microcontroller relative to the board ground, while the SAM4L microcontroller's total current consumption is measured through available measurement pins on the board. Time stamped data is logged with a locally developed LabVIEW program, and used together with our voltage and current measurements, to find the energy consumption of the H.264/AVC encoder control structure. The single-core FSS framework is used to implement the H.264/AVC encoder control structure with system scenarios.

Our version of the SAM4L board contains the ATSAM4LC4CA microcontroller. The microcontroller contains an ARM Cortex M4 processor, and supports two different run modes. RUN0 operates with a core voltage of 1.8 V, while RUN1 operates at 1.2 V. Table 3.5 shows measured power consumption for the two run modes under heavy load and in idle condition. The microcontroller also supports various low-power states. Their possible use is described in the RTH (Run-to-Hold) Section of Chap. 4.

For the scope of our research, we have selected the internal oscillators RCFAST (12 MHz) and RC80M (80 MHz) as the main clock sources for our DVFS settings.

Table 3.5 Run modes in our SAM4L-board from Atmel

Mode	Measured power consumption (μW)
RUN0 (heavy load)	29,865
RUN0 (idle)	25,340
RUN1 (heavy load)	5107
RUN1 (idle)	4013

Since the ARM Cortex M4 processor only supports frequencies up to 48 MHz, the RC80M oscillator is clock divided to 40 MHz when it is selected as the main clock source. Hence, our DVFS settings are PS0 (1.8 V and 40 MHz) and PS1 (1.2 V and 12 MHz).

Based on the SAM4L board datasheet, the DVFS is triggered by a write to configuration registers after a write to protective lock registers. In other words, a switch is triggered after a few clock cycles. We have measured the switching time between our two clock settings with an oscilloscope. The switch takes approximately 15 clock cycles (at 12 MHz) when both oscillators are on continuously, and approximately 50 clock cycles (at 12 MHz) if the currently unused oscillator is powered down. The added power consumption by having both oscillators enabled is negligible because unused oscillators that are idling does not contribute to any measurable power consumption. Therefore, both oscillators are kept enabled in our measurements for a significantly reduced switching time.

We have measured the total static power to be less than 16% of the total power consumption in PS0, and less than 29% in PS1. The static power consumption on the SAM4L board is approximated by measuring the total power consumption for two different frequencies in each power mode. With these four equations we estimate the effective capacitance and static energies under the assumption that the activity factor (α) is equal to 0.5 in Eq. (3.8).

$$P_{\text{Total}} = P_{\text{Static}} + P_{\text{Dynamic}} = P_{\text{Static}} + \alpha C V^2 f \tag{3.8}$$

When performing dynamic voltage scaling on the SAM4L board, the main clock frequency supported in the new voltage level has to be configured first. According to the datasheet, the 1.2 V (PS1) configuration does not support frequencies larger than 12 MHz. The frequency setting must therefore be switched from 40 MHz to 12 MHz before the voltage switch from 1.8 V (PS0) to 1.2 V (PS1). When changing configuration from PS1 to PS0 the frequency must be scaled after the voltage switch to maintain stability.

In addition, the output from the voltage regulator has to be stable after a voltage switch before the CPU can continue processing. After a switch from PS1 to PS0 we have to wait for a status flag, which signals that the voltage level has stabilized. The CPU is therefore stalled while waiting for this flag to be set in the SAM4L board. However, when a voltage switch is triggered from PS0 to PS1, then waiting for this flag is not necessary before continuing the CPU execution. As described by Atmel Support, when switching from a higher voltage level to a lower level, the voltage regulator is stable and will regulate fast enough to continue execution in succeeding cycles. The measured current consumption will gradually decrease after this switch until the charge on the decoupling capacitor connected to the core voltage has been consumed down to its PS1 level. The power delivered from the regulator on the other hand is equal to the PS1 level immediately after the voltage switch.

Our SAM4L board came initially with a 22 μF decoupling capacitor connected between the core voltage input and ground. The large decoupling capacitor resulted in extremely long switching times. This was confirmed by Atmel Support, and they

Table 3.6 DVFS switching times between PS0 and PS1

From	To	Cycles @ 12 MHz	Switching time (μS)	Overhead energy (μJ)
PS1	PS0	197	16.40	0.0623
PS0	PS1	15	1.25	0.0281

recommended a change to 4.7 μF to guarantee that all modules would function properly. Since we are only interested in the CPU, we measured the DVFS switching times with lower decoupling capacitors. Our experiments showed that the CPU functioned correctly using a 1 μF decoupling capacitor, giving a DVFS switching time comparable to state-of-the-art processors. Table 3.6 shows the switching times and energy overhead when switching from PS0 to PS1 and vice versa. The energy overhead is found by measuring the time between the last instruction before and the first instruction after the DVFS switch, before multiplying this time with the power consumption for the run mode before the switch. An oscilloscope was used for the timing measurement.

In our experiments we simulate a realistic video sequence of 500 s, assuming the video sequence can be processed with 25 ARM Cortex M4 cores in parallel in order to meet the timing requirements for a frame rate of 25 frames per second (fps). The parallel cores are assumed able to process each frame individually with negligible synchronization overhead. This solution requires buffering and software pipelining of the frames to maintain forward frame dependency requirements. Other researchers have suggested similar approaches where their focus targets the parallelization of the H.264/AVC encoder [15, 18]. We do not target general high performance multi-core platforms. Instead, we aim at a subset of applications with non-interacting concurrently operating tasks. Moreover, we target the embedded multi-core domain, where cache coherence and complex bus protocols are avoided by limiting the functionality of the cores. In particular, we limit ourselves to totally independent cores working on tasks that do not have any interaction, where we believe this extrapolation is valid.

Our scenario implementation is independent of this parallelization, and can be used on a faster processor for a single-core solution. After each frame is processed, we force the core into a low-power retention mode, where the measured energy consumption is negligible. Before entering retention mode, a configurable timer is set to wake up the system just before the next frame arrives.

Our video sequence is profiled through measurements on three different wireless networks; a wireless local area network (WIFI), a mobile long-term evolution (LTE) network, and a wideband code-division multiple-access (WCDMA) network, as shown in Fig. 3.12 [16]. To process a frame, we specify a threshold requirement, bw_th, for the available bandwidth to be equal to at least five times (5×) the required bandwidth for a frame rate of 25 fps with a particular frame size. This threshold allows other activity to continue on the network, and avoids using up all the available bandwidth. With lower thresholds, the limited number of available voltage levels in the SAM4L board would always force the system to use the most power

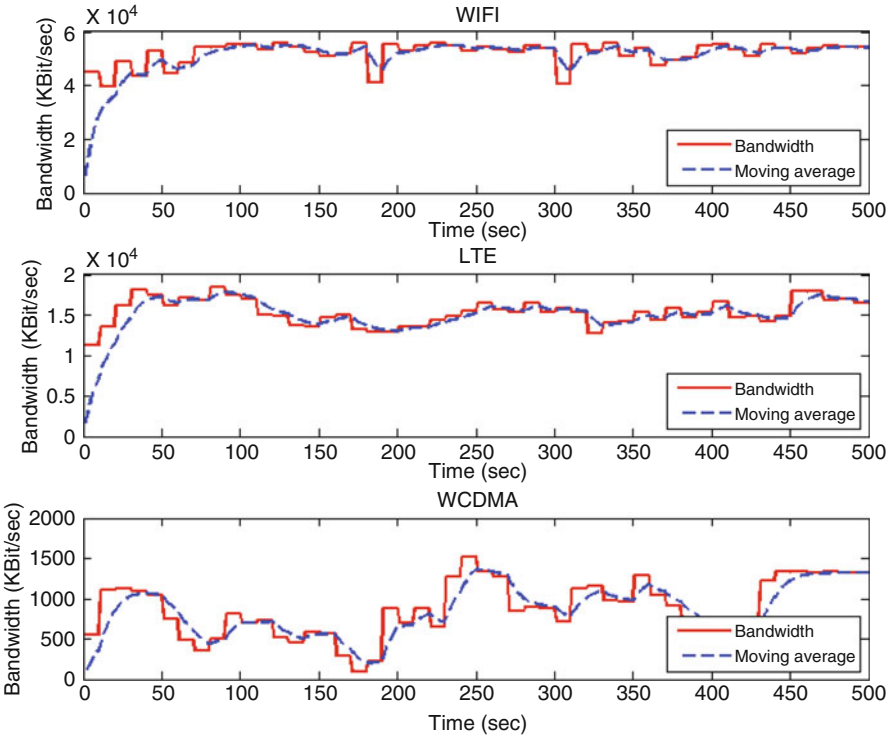


Fig. 3.12 Available bandwidth measurement and moving average over WIFI, LTE, and WCDMA [16]

consuming voltage setting. The 5× threshold thus enables us to demonstrate how our methodology exploits the dynamism in the available bandwidth. Our simulated 500 s video sequence is created by combining different frame sizes from Table 3.2 in a sequence that fits with the available bandwidth of the measured networks in Fig. 3.12, and our threshold requirements.

The application code is compiled in a conventional way with compiler optimization parameter O3. When we optimize the application code with the O3 configuration, we assume it is a good enough optimization for the exploitation step in the system scenario design methodology. The energy consumption is measured in real time using our experimental setup, with and without the scenario mechanism. The overhead energy consumption of our scenario mechanism is measured by forcing the scenario mechanism to select PS0 each time a scenario switch takes place. This approach would not reduce the energy, but would trigger the scenario mechanisms in our AMU and PAM modules. The results are then compared to the energy consumption without using the scenario mechanism. The difference equals the scenario mechanism overhead.

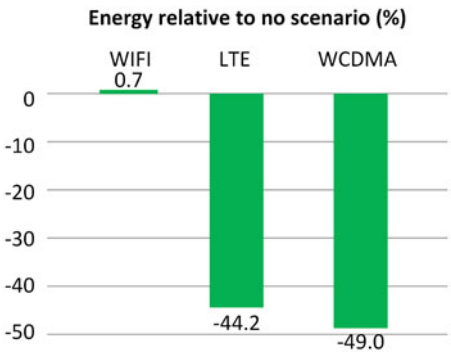


Fig. 3.13 Energy consumption relative to no system scenarios

6.2 Discussion of Obtained Results

Figure 3.13 shows the energy reductions relative to the energy consumed by a no scenario implementation utilizing a race-to-halt (RTH) setup in each of the measured networks of Fig. 3.12. A retention mode is used when the system is halted, which consumes 4.4μW. The average energy overhead of our scenario mechanisms is measured to be 0.7%, which we consider to be negligible.

The lowest energy-consuming situation occurs when all frame sizes are suitable for the PS1 configurations, such as for the WCDMA network. We obtain in this situation up to 49% energy reduction as compared to a RTH solution without the use of the scenario mechanism. These results include the average scenario mechanism overhead of 0.7% energy increase for the 500 s sequence.

When the available bandwidth is higher than our threshold for our largest frame size, then the scenario mechanism is not needed. It simulates our most energy-consuming situation, where the system is running constantly in the PS0 platform configuration. In this situation the energy consumption does not increase more than the measured scenario overhead.

The LTE results are interesting because the energy reduction for this network is determined by how long the system is in the PS1 configuration. In our measurement, the platform runs with the PS1 configuration 93.8% of the time for this LTE network measurement (with bw_th = 5×), and runs in the PS0 for the remaining sequence. The energy savings would decrease if the platform runs in PS0 for longer time periods.

7 Conclusions

In situations where many-valued data variables influence the dynamic behavior of an application, techniques developed for control variables using multi-valued decision diagrams explode with respect to run-time and memory complexity.

Alternatives techniques, e.g., based on polyhedral parameter space partitioning and RTS detection, need to be employed. In addition, efficient techniques for application monitoring and platform adaption are required. When this is done, significant improvements in performance and energy reduction can be achieved, compared to implementations that do not take data variable based dynamism into account through a system scenario methodology.

Acknowledgements The research leading to these results has in part been performed within the context of the dual-PhD agreement between KU Leuven and NTNU. Furthermore, the authors would like to thank Associate Professor Sverre Hendseth at NTNU for his many contributions to the research.

References

1. M. Ashouei et al., A voltage-scalable biomedical signal processor running ECG using 13 pj/cycle at 1 mhz and 0.4 v, in *Proceedings IEEE International Solid-State Circuits Conference (ISSCC)* (2011), pp. 332–334
2. M.J. Atallah, M. Blanton (eds.), *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*, 2nd edn. (Chapman & Hall/CRC, New York, 2010)
3. Atmel, SAM4L Xplained Pro user guide (2014)
4. M. Baka, F. Catthoor, D. Soudris, Proposed evaluation framework for exploration of smart PV module topologies, in *European Photovoltaic Solar Energy Conference (PVSEC)*, Munich, Germany (2016), pp. 176–179
5. B. Chazelle, J. Friedman, Point location among hyperplanes and unidirectional ray-shooting. Comput. Geom. **4**(2), 53–62 (1994)
6. I. Filippopoulos, F. Catthoor, P.G. Kjeldsberg, Exploration of energy efficient memory organisations for dynamic multimedia applications using system scenarios, Des. Autom. Embed. Syst. **17**(3–4), 669–692 (2013)
7. E. Hammari, F. Catthoor, P.G. Kjeldsberg, J. Huisken, K. Tsakalis and L. Iasemidis, Identifying data-dependent system scenarios in a dynamic embedded system, in *The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'12)*, Las Vegas, USA (2012)
8. E. Hammari, P.G. Kjeldsberg, F. Catthoor, Run-time precomputation of data-dependent parameters in embedded systems. ACM Trans. Embed. Comput. Syst. **17**(3), Article No. 68 (2018)
9. L. Iasemidis, Seizure prediction and its applications. Neuroimaging Clin. N. Am. **22**, 489–506 (2011)
10. L. Iasemidis et al., Long-term prospective on-line real-time seizure prediction. Clin. Neuropathol. **116**, 532–544 (2005)
11. M. Kallay, The complexity of incremental convex hull algorithms in R^d. Inf. Process. Lett. **19**(4), 197 (1984)
12. C. Lee et al., MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, in *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture* (1997), pp. 330–335
13. Z. Ma et al., *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogenous Platforms* (Berlin, Springer Publishing Company, 2007), ISBN 978-1-4020-6328-2
14. National Instruments, NI myDAQ measurement board, http://www.ni.com/mydaq on December 7, 2015

15. S. Sun, D. Wang, S. Chen, A highly efficient parallel algorithm for H.264 encoder based on macro-block region partition, in *High Performance Computing and Communications. Lecture Notes in Computer Science*, vol. 4782 (2007)
16. Y. Yassin, P.G. Kjeldsberg, A. Perkis, F. Catthoor, Dynamic hardware management of the H264/AVC encoder control structure using a framework for system scenarios, in *Euromicro Conference on Digital System Design, DSD 2016, Limassol, Cyprus* (August, September 2016)
17. Y. Yassin, P.G. Kjeldsberg, A. Perkis, F. Catthoor, Techniques for dynamic hardware management of streaming media applications using a framework for system scenarios, Microprocess. Microsyst. **56**, 157–168 (2018). https://doi.org/10.1016/j.micpro.2017.12.002
18. Z. Zhao, P. Liang, A highly efficient parallel algorithm for H.264 video encoder, in *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, Toulouse, France (May 2006)

Chapter 4
DVFS-Oriented Scenario Applications to Processor Architectures

Nikolaos Zombakis, Yahya H. Yassin, Michail Noltsis, Dimitrios Soudris, Per Gunnar Kjeldsberg, and Francky Catthoor

1 Software-Oriented Applications

As the control variable oriented scenario methodology is quite general [8], we have exploited this approach in the context of DVFS mapping on instruction-set programmable processors. As shown in [6, 7] an automatic trajectory which has been developed by Valentin Gheorghita, e.g., for a specific design problem, namely the identification, prediction, and exploitation of application scenarios to reduce the energy consumed by a single task streaming application running on a dynamic voltage and frequency scaling (DVFS) aware processor. DVFS is an effective energy-saving technique that consists of varying the frequency and voltage

N. Zombakis (✉) · D. Soudris
National Technical University of Athens, Athens, Greece
e-mail: nzompakis@microlab.ntua.gr; dsoudris@microlab.ntua.gr

Y. H. Yassin
Norwegian University of Science and Technology, NTNU, Trondheim, Norway

KU Leuven, Leuven, Belgium

M. Noltsis
National Technical University of Athens, Athens, Greece

KU Leuven, Leuven, Belgium
e-mail: mnoltsis@microlab.ntua.gr

P. G. Kjeldsberg
Norwegian University of Science and Technology, NTNU, Trondheim, Norway
e-mail: pgk@ntnu.no

F. Catthoor
IMEC and KU Leuven, Leuven, Belgium
e-mail: catthoor@imec.be

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6_4

of a processor at run-time according to processing needs. The exploited algorithms are general and they may be used for different problems for both hard and soft real-time constraints. It starts from an application written in C, as C is the most used language to write embedded systems software, and generates the final energy-aware implementation also in C. Promising results have been obtained for the hard and soft real-time setups, both for an MP3 Decoder and a Motion Compensation Video Kernel. For the latter kernel, under hard real-time constraints, a saving has been achieved of up to 69% using a coarse-grain scheduler and up to 75% when using a fine-grain scheduler, when comparing with the reference implementation. In the soft real-time context, the savings were up to 16%, which represents about 50% of the theoretically possible reduction.

The basic control variable scenario-based DVFS techniques have been further exploited in the context of real-time wireless baseband systems. Also there promising results have been obtained as elaborated in [13].

2 DVFS-RTH Sleep Mode Extensions

Race-to-halt (RTH) [3] is a state-of-the-art technique for energy saving, speeding up the execution of an application, thus increasing instantaneous power consumption, in order to finish early and achieve longer idle times. The static power consumed during idle times is reduced significantly due to the utilization of extremely low-power states. A processor may have more than one low-power state, or sleep mode, each having a different energy-saving potential but also a different cost related to going to sleep and waking up. Selecting the optimal sleep mode is then a challenging task, given that you typically have limited information about the length of the idle period. In this section, we show how sleep modes can be efficiently combined with DVFS and system scenarios in situations where we have a limited set of voltage and frequency alternatives.

2.1 Sleep Mode Management

In [3] Awan et al. present a technique for sleep mode selection. It is based on an offline and online approach, where the most efficient sleep modes are found offline for a set of jobs based on the maximum time interval, t_1, for which the processor may be forced into a sleep state without causing any task to miss its deadline under worst-case assumptions. The online algorithms compute the sleep time by calculating the available slack present at run-time. The most efficient sleep state is then selected online based on the sleep time calculated online and the pre-calculated t_1 interval.

The system scenario approach enables us to simplify the solution in [3] significantly. Instead of computing the slack online, the total number of available cycles for each scenario is calculated offline based on what we know about our system from

Table 4.1 Run/sleep modes in our SAM4L-board from Atmel

Run/sleep mode	Measured power consumption (μW)	Wakeup time (incl. setup time)
RUN0 (heavy load)	29,865	N/A
RUN0 (idle)	25,340	N/A
sleep00	14,010	83 clk @ 40 MHz
sleep01	10,370	659 clk @ 40 MHz
sleep02	5460	664 clk @ 40 MHz
sleep03	79	3907 clk @ 40 MHz
wait0	12	5669 clk @ 40 MHz
retention0	5.8	5997 clk @ 40 MHz
RUN1 (heavy load)	5107	N/A
RUN1 (idle)	4013	N/A
sleep10	2440	88 clk @ 12 MHz
sleep11	1830	390 clk @ 12 MHz
sleep12	1460	395 clk @ 12 MHz
sleep13	51	1057 clk @ 12 MHz
wait1	5.9	1728 clk @ 12 MHz
retention1	4.4	6703 clk @ 40 MHz 1786 clk @ 12 MHz

the scenario identification step described in Chaps. 2 and 3. For a given platform, we also have stored system wakeup times from each of the possible sleep modes. The SAM4L board described in Chap. 3, for instance, contains different sleep modes as shown in Table 4.1. In Listing 4.1 the total number of available frame cycles (TOTAL_FRAME_CYCLES) is pre-calculated offline for each scenario (PS0 and PS1 in this case) based on information from the scenario identification step. The stored wakeup time, including the setup time, is then summed with the calculated loopcnt value (g_rth_lc and g_rts_data in Listing 4.1) and compared to the pre-calculated total frame cycles available online. Our solution then selects the lowest power consuming sleep mode if our estimated total execution time is less than the slack time available. The different sleep power consumptions, setup times, and wakeup times are measured on our SAM4L board, and included in Table 4.1. All these wakeup time values (including the setup time) are stored in the $grthlc$ global lookup table. This lookup table is read by the AMU_2 mechanism shown in Listing 4.1.

AMU_2 reduces the online overhead complexity from Awan et al. to an if-else comparison. Starting with the lowest power consuming sleep mode, the AMU_2 compares the slack available with the sum of loopcnt and the corresponding sleep wakeup time until it finds a solution. If no solution is found the system stays idle until the arrival of the next frame. Chapter 3 contains two code listings that should be seen in relation to Listing 4.1. The first one contains the AMU_1 inline function that selects a DVFS voltage and frequency according to the detected scenario.

```
static inline void AMU_2(volatile int *scenario){
  if(*scenario == DVFS_40MHZ_1_8V){ // Current scenario = PS0
    if( ((g_rts_data)[0] + (g_rth_lc)[5]) < TOTAL_FRAME_CYCLES_40MHZ )
      (g_rts_data)[1] = (long) PS0_RET1; // loopcnt + RET1 wakeup < Slack
    else if( ((g_rts_data)[0] + (g_rth_lc)[4]) < TOTAL_FRAME_CYCLES_40MHZ )
      (g_rts_data)[1] = (long) PS0_RET0; // loopcnt + RET0 wakeup < Slack
//...
    else if( ((g_rts_data)[0] + (g_rth_lc)[0]) < TOTAL_FRAME_CYCLES_40MHZ )
      (g_rts_data)[1] = (long) PS0_SLEEP00; // loopcnt + SLEEP00 wakeup <
          Slack
    else
      (g_rts_data)[1] = (long) PS0_RUN0; // No RTH
  }else if(*scenario == DVFS_12MHZ_1_2V)\{ // Current scenario = PS1
    if( ((g_rts_data)[0] + (g_rth_lc)[10]) < TOTAL_FRAME_CYCLES_12MHZ )
      (g_rts_data)[1] = (long) PS1_RET1; // loopcnt + RET1 wakeup < Slack
//...
    else if( ((g_rts_data)[0] + (g_rth_lc)[6]) < TOTAL_FRAME_CYCLES_12MHZ  )
      (g_rts_data)[1] = (long) PS1_SLEEP10; // loopcnt + SLEEP10 wakeup <
          Slack
    else
      (g_rts_data)[1] = (long) PS1_RUN1; // No RTH
  }
}
```

Listing 4.1 AMU_2 structure

Afterwards the the nested loops of the next listing are executed, which is then again followed by the AMU_2 inline function in Listing 4.1.

2.2 Sleep Mode Experimental Results

The application section of Chap. 3 has shown experimental results for the H.264/AVC video encoder [9] control structure run on an SAM4L board from Atmel [2]. Profiles of three wireless networks (WIFI, LTE, and WCDMA) with varying bandwidth were combined with a video stream, where the resolution was dynamically adapted to the current bandwidth. The energy overhead of our sleep mode enabled scenario mechanism is now measured using this stream with and without the AMU_2 and its corresponding lookup tables for the system scenario sleep mode management. The lookup tables for the scenario sleep mode management are based on Tables 4.1 and 4.2. The added overhead of the AMU_2 is measured to be less than 0.1% and therefore we consider the added overhead negligible. With our experimental setup all frames, even the largest one, are processed and finished with enough slack time for our AMU_2 to select the lowest power consuming sleep mode. The sleep mode scenario consequently does not give any energy reduction. If, on the other hand, the slack time of different frame sizes would be in the same range as wakeup times, we would benefit from selecting the optimal sleep mode. To illustrate this claim, we have simulated our recorded streams assuming the wakeup time of each sleep mode was a factor N slower, where N equals 1×, 10×, 100×, 1.000×, 10.000×, 20.000×, and 66.973×. If we were

Table 4.2 DVFS switching times between PS0 and PS1

From	To	Cycles @ 12 MHz	Switching time (μS)	Overhead energy (μJ)
PS1	PS0	197	16.40	0.0623
PS0	PS1	15	1.25	0.0281

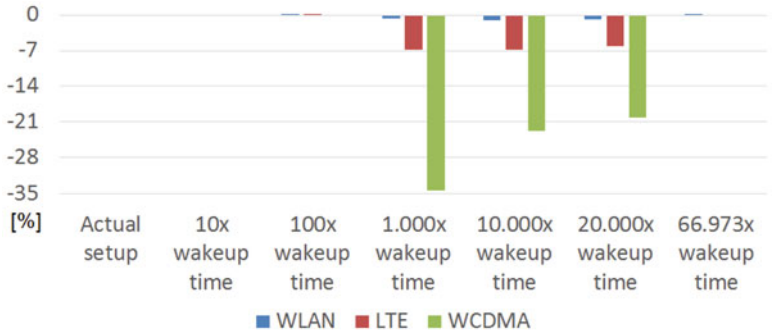


Fig. 4.1 Potential sleep scenario improvements in energy consumption (in percent) with slower wakeup times, compared to race-to-halt (RTH) scenario without sleep mode selection

actively using other types of memories (e.g., off-chip memories), then the wakeup times for these memories are more likely to be at least 1.000 times slower, which makes our assumptions realistic. If the slack time would be significantly shorter than what we have in our experiment, such that our most power efficient sleep mode could not be selected, then we also expect similar energy reductions. If the wakeup time is too long, which is the case if $N > 66.973\times$, the system would never be able to go into a sleep mode. Based on this we see that our application benefits from the AMU_2 if its task slack times dynamically change between different processing intervals and between different sleep mode wakeup times.

Without AMU_2 the RTH mechanism cannot adapt the sleep mode selection according to dynamically changing slack times. Instead the least power consuming sleep mode that still is able to wake up in time after the largest frame size has to be selected in all situations. This sleep mode will be increasingly more power consuming when N increases, and for $N > 66.973\times$ the wakeup times of all sleep modes will be too long. The AMU_2 mechanism exploits lower power consuming sleep modes when smaller frames are processed, instead of only selecting the safest sleep mode. The estimated energy reductions compared to not having the AMU_2 (and corresponding lookup tables) available are shown in Fig. 4.1. Figure 4.2 shows the energy reductions compared to a race-to-idle (RTI) setup where the processor does not go to sleep or halt after each processed frame.

Figures 4.1 and 4.2 clearly show that we would start to get significant energy reductions if the wakeup times for our sleep modes were approximately 1.000 times slower. In this case, our AMU_2 approach would result in up to 34.3% energy reduction. The effect decreases when N increases since fewer sleep modes are within the range of the slack times. In our simulation, we observe that if the wakeup times

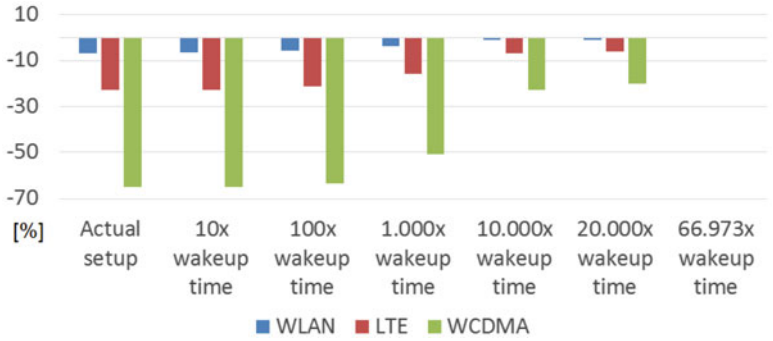


Fig. 4.2 Potential sleep scenario improvements in energy consumption (in percent) with slower wakeup times, compared to scenario with race-to-idle (RTI)

were more than 66.973 times slower, then even the sleep mode with the shortest wakeup time would become too slow. In this situation our processor would stay idle until the next set of frames arrive. When the wakeup times are only 100 times slower, we observe a slight increase in energy consumption. This effect is caused by the fact that our AMU_2 mechanism selects sleep mode sleep03 over sleep02 because it has lower power consumption. On the other hand, the sleep03 wakeup time is significantly larger than the sleep02 wakeup time as shown in Table 4.1, and this large difference affects the total consumed energy consumption. If the wakeup times would be more evenly scaled between the different sleep modes, then this effect would not appear.

3 Reliability-Sensitive Hardware-Oriented Applications and Gas-Pedal Extension

The target of this subsection is to present a DVFS extension using system scenarios that mitigates performance variability through the use of a PID controller [11]. The timing guarantees are managed via the use of some *gas-pedal*, DVFS points of elevated voltage and frequency from custom DVFS ones. These points are generally "out-of-specs" and are verified through experiments of the pure hardware. This way, the system can steal slack delay by operating for short periods in this "speed-up" mode. *Therefore, this extension is tightly tied to "any" dynamic DVFS control approach, in particular when it is coupled with the use of system scenario exploitation and prediction.*

In fact, dependability has always been a major concern for the design of digital systems. Modern computing has to meet reliability specifications, while the dimensions of semiconductor devices reach tens of nanometers. This downscaling trend brings about a variety of mechanisms that cause fluctuations of operating

parameters of digital systems. To mitigate the numerous reliability threats, computer engineers have developed a plethora of RAS (reliability, availability, and serviceability) techniques focusing on different steps of the fabrication process, from design to post-fabrication stages. We could classify these mechanisms between two major groups: the ones targeting *functional reliability* violations while establishing binary correctness and the ones focusing on *parametric reliability*, mitigating the fluctuation of performance parameters that deviate from expected system behavior.

3.1 Performance Dependability

As a case study, the controller has been instantiated on the NXP IMX6Q board [12] and developed on top of the Spectrum Sensing embedded application [4], a realistic application with hard deadline constraints. To control performance, the PID principle is to switch frequency and voltage of the processor based on a reactive response to monitor timing delays. To this end, we have introduced the definition of *slack* as the timing difference for specific code chunks, between their execution under nominal conditions and their execution under timing violations.

Firstly, our scheme aims in managing *functional reliability*. Therefore, errors need to be detected and recovered. Having an error-detection mechanism has been part of our previous work [5]. Briefly, a *signature-based* strategy is used that alters the signature trace progressively using internal application data. At regular time intervals, the trace is compared against a golden signature that has been generated offline at error-free conditions and in the case of a mismatch, the error is detected. It should be underlined that through this configuration, error identification happens at run-time, within 1 ms from their occurrence. It is also important to note that signature-based detection cannot detect errors in all application data structures nor can identify the source of the error in the processor circuitry.

Significant part of our work has been the development of an error-recovery mechanism that ensures correct system functionality. Hence, we developed a rollback technique to correct the detected errors. To elaborate in a more detailed manner, after an error has been detected, the system is rolled back to a previous known state, where the operation is repeated. It is evident that this procedure introduces extra clock cycles and thus timing delay. It should also be highlighted that with such a rollback event, the system can recover only from transient (or recurring) errors; in the presence of permanent or semi-permanent errors, correct system operation cannot be ensured as such. Then we have to add fault-tolerant architecture features in order to start up/power up redundant architecture resources when the initial ones become unusable due to the permanent faults. Figure 4.3 summarizes our developed RAS mechanism.

Core of the PID implementation is the monitor used to measure timing delay and estimate *slack* along with the knobs used to switch to other DVFS points. For monitoring purposes, we use the `clock_gettime()` function of the *C* library that allows measuring time intervals between code chunks with a millisecond

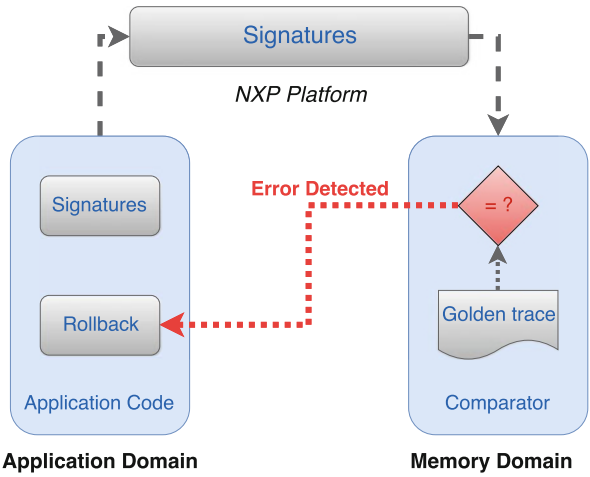


Fig. 4.3 The configuration of our RAS mechanism. At certain checkpoints of the application, the system computes the signature and compares it against the golden trace. In case of a mismatch, a rollback event occurs to recover correct operation

accuracy. Another way to calculate function time is by reading cycle counters (`CCNT`). By default, these counters are disabled so they have to be enabled from kernel-mode. In our processor, this counter is accessible via CP15 instructions. DVFS is supported on our target board, though default support is minimal. Three DVFS points are present, corresponding to minimum, nominal, and maximum frequency. This restriction however is a software limitation, and not a hardware one.

To this end, we used a custom DVFS driver [4] that enables the use of numerous operating points (up to 50). In this way we can achieve a fine-grain granularity of the frequency and voltage space. After thorough experimentation with the board's cores, we have managed to link each frequency step to its relevant voltage, based on the following mathematical formulation:

$$V = 9725 + 5 \times (F - F_{\min}) \qquad \text{(in 100 uV steps)} \qquad (4.1)$$

where $F_{\min}$ represents the minimum frequency of our board 396 MHz and voltage is in 100 μV steps (9725 = 0.9725 V). It should also be stressed that frequency is rounded to 12 MHz increments, while voltage is converted to 1 mV steps.

3.2 *Introducing Gas-Pedal Points*

We should highlight the use of one additional *gas-pedal* point—elevated both in terms of voltage and frequency—from custom DVFS ones. This point has also been verified through experimental tests. Specifically, we have observed that functional stability is not harassed when the processor is operating at such an out-of-spec level for a certain time window. Therefore, we select to use this "turbo" mode on our

scheme and let the processor function at such levels for small periods of time. Of course, in general, the introduction of gas-pedal points is not limited to the use of solely one point. In other cases, more such points can be utilized nevertheless, for our scheme we have identified only one.

3.3 Choosing the Operating Points

To decide the final number of DVFS points to use for our configuration we experimented with the controller, instantiating different number of operating points, equally placed between maximum and minimum frequency limits. Each time we injected an error, provoking a rollback event and measured the settling time for all different configurations, that is, the time needed for *slack* to converge to zero and remain within a reasonable error band (in our case few milliseconds). Figure 4.4 sums up our experimental results.

As can be seen, increasing the DVFS points greatly reduces settling time at first. After a certain point, however, a further increment barely reduces settling time. It should also be noted that in practice, it is unusual for embedded processors to offer a pool of numerous operating points. Lastly, avoiding the use of many different DVFS points protects the controller output from jittering, along with the energy and timing overhead stemming from excessive DVFS switches. For the above reasons, we concluded to utilize nearly a dozen operating points for our configuration as shown in Table 4.3. Notice the final "*gas-pedal*" point.

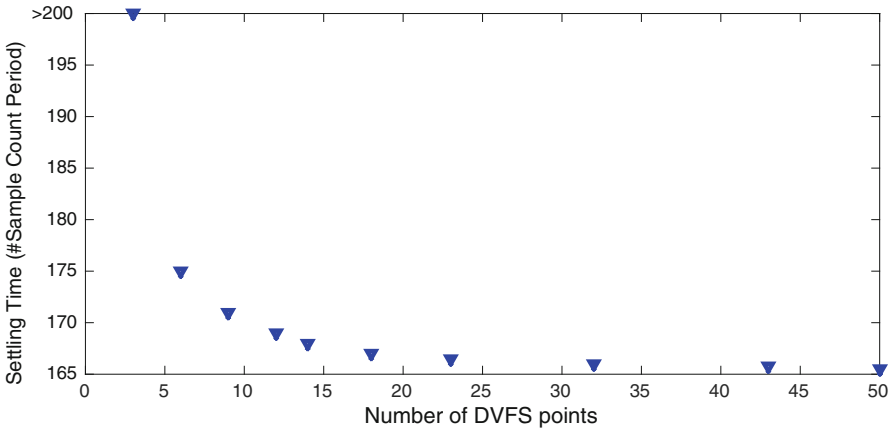


Fig. 4.4 Graph showing the dependence of settling time on number of DVFS points. Settling time per each configuration is the mean value as estimated after 10 iterations

Table 4.3 Operating points on our processor

Available DVFS levels		
m	f (MHz)	V_{dd} (V)
0.50	396	0.975
0.56	444	1.00
0.62	492	1.025
0.68	540	1.050
0.74	588	1.075
0.80	636	1.100
0.88	696	1.125
0.94	744	1.150
1.00	792	1.175
1.06	840	1.200
1.12	888	1.225
1.20	946	1.250
1.26	996	1.275
1.32	1044	1.350

3.4 Case-Study Experiments

After instantiating the controller, we proceed with our case study to examine the capabilities of our proposed scheme. We should underline that the NXP board is fabricated at 40 nm technology nodes from TSMC [10] while the processor consists of an ARM Cortex-A9 [1] quad core. Furthermore, our case study consists of a commercial application that performs frequency band characterization and allocation with heavy signal processing computations. Nevertheless, our proposed scheme can seamlessly be integrated on top of any application which meets the target application domain assumptions.

3.4.1 Dependability in the Presence of Rollback Interventions

For the experiments, we maintain the nominal frequency at 792 MHz and voltage at 1.175 V of the chip. First, we collect the golden trace at nominal, error-free conditions. For a qualitative realization of the controller's power consumption we refer to normalized power, assuming power scales quadratically with voltage ($P \propto V_{dd}^2$). We should underline that our experiments are performed on a low-power processor without heavily scaled nodes; therefore, leakage can still be neglected. Hence, we have not incorporated that part in our energy calculations. However, our approach can be ported also to such scaled nodes and leakage should of course be included. By proper power gating our dynamically triggered rollback based mitigation approach should not lead to issues linked to a large leakage overhead.

In Fig. 4.5 we can observe how the *slack* measurement is progressing over time. In our case, time is viewed as sample count period, that is, the time for

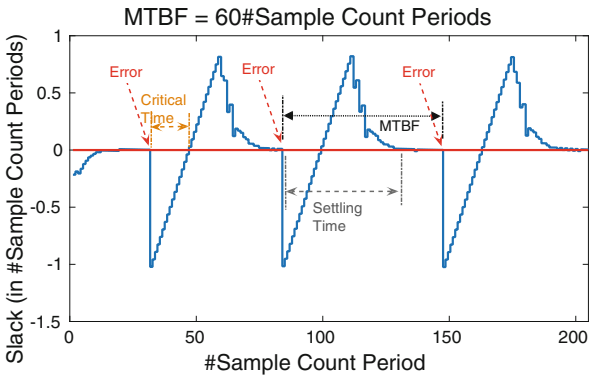


Fig. 4.5 Slack versus time in the case of rollback interventions

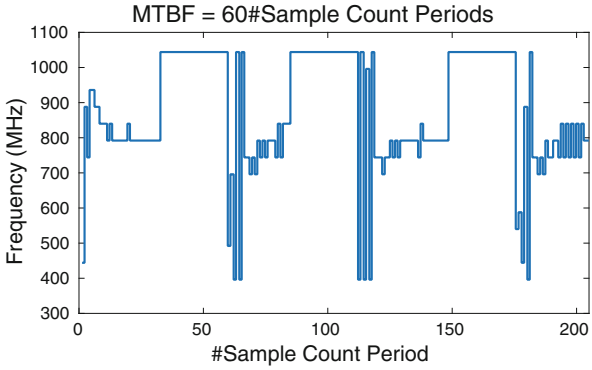


Fig. 4.6 Frequency decisions versus time in the case of rollback interventions

the application to perform one basic data processing computation at nominal frequency.[1] When an error occurs, a signature mismatch is detected and almost immediately a rollback event takes place. This introduces delay, leading *slack* to drop to negative values. Afterwards, the controller takes action, boosting frequency to absorb timing overheads and converge *slack* to zero.

In Fig. 4.6 we see the frequency decisions of the controller. As expected, the controller drives frequency to its maximum level just after the rollback event and when the settling time has passed, *slack* converges to zero—thus meeting our deadlines—while frequency remains near its reference value. Notice that the selection of the frequency steps is a function of the error rate, the controller gains, and the granularity of the DVFS points. Different rates of errors or different tuning of PID gains can activate different groups of DVFS points. Hence, the presented results refer to an instance of the possible scenarios.

[1]Target board is low-power oriented; hence, this basic computational block is measured to last significantly longer than in a high performance platform.

3.4.2 Dependability in the Presence of Extra Load

We move forward by assessing the controller's response to guarantee dependability at the presence of workload variation while the board operates at nominal, error-free conditions. In this case, workload variation occurs when another application is running on the same core, competing for logic and memory resources. Hence, the challenge for our controller is to compensate for the extra load by optimal DVFS responses.

The imposed *parametric* reliability threat is visible at Fig. 4.7. We can identify the timing instance where extra load is applied to the CPU when sudden *slack* drops appear. Then, the controller boosts the processor's frequency and converges *slack* to zero. Positive *slack* is expected when the extra load is removed and in this case, the processor slows down since meeting the timing deadlines can be achieved at lower frequencies and power budgets. Figure 4.8 shows the frequency response of the controller.

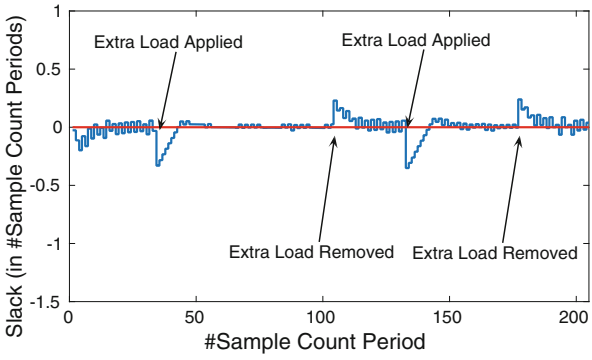


Fig. 4.7 Slack versus time in the case of extra workload

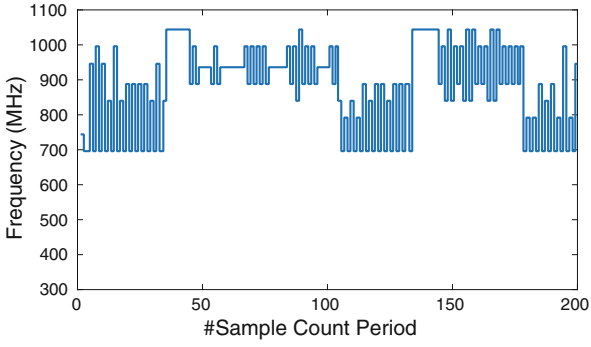


Fig. 4.8 Frequency decisions versus time in the case of extra workload

3.5 Hardware-Related Limitations of Our Scheme

Although our scheme is able to provide dependability guarantees, it is important to note the hardware-related limitations. These limitations are directly correlated to the hardware's maximum operating frequency; the reason is obvious: a higher operating frequency is able to absorb timing overheads imposed by RAS events or extra workload faster. Such an approach can be visible by focusing on Fig. 4.5 and noticing *critical time* T_{crit}. We define as T_{crit} the time needed for the processor to reach zero *slack* after a rollback event. This is in contrast to *settling time* T_{set} from control theory, which represents time needed for *slack* to converge to zero (remain in zero within a reasonable/small margin).

For our processor, the lowest $MTBF$ rate the controller can handle is shown in Fig. 4.9. For this error rate, controller's decision is to force frequency to the highest step as seen in Fig. 4.10. As expected, power is significantly higher compared to previous cases because of the maximum voltage operation. For an error rate higher than the aforementioned ($MTBF < T_{crit}$), timing deadlines cannot be met. An example of such a case is shown in Fig. 4.11. This of course is a limitation of

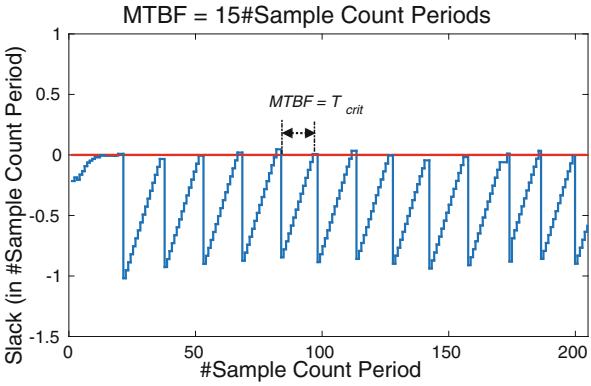


Fig. 4.9 Slack versus time for $MTBF = T_{crit}$

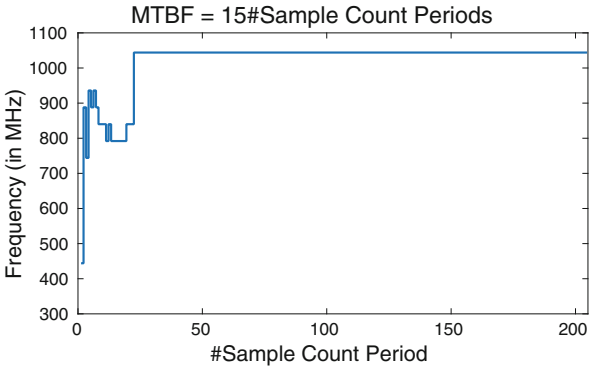


Fig. 4.10 Frequency decisions versus time for $MTBF = T_{crit}$

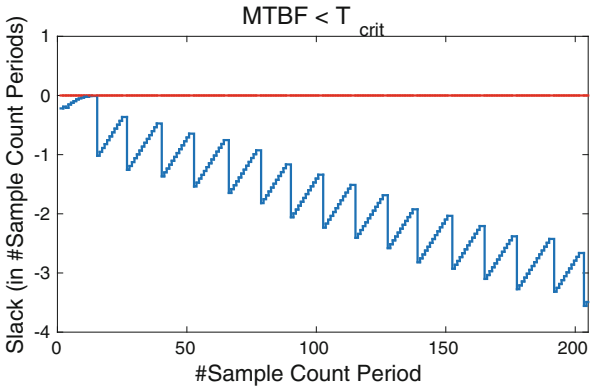


Fig. 4.11 Slack versus time for $MTBF < T_{crit}$

the hardware and not the controller. Should a higher frequency step be available, the controller could utilize it and absorb even more timing delays during the operation.[2]

4 Conclusions

In this chapter, we have introduced our projections of the generic system scenario methodology to the DVFS based scheduling context. First, we have exploited this for DVFS-aware scheduling in both hard and soft real-time systems. Then, we have added a novel sleep-mode extension which is especially useful for embedded systems that have low duty cycles and that hence exhibit long periods of power down. Finally, we have introduced a PID controller mechanism realized on top of a DVFS scheduler that mitigates performance variability in the context of RAS mechanisms. The basic techniques are discussed in Chap. 7. But the underlying DVFS scheduler is summarized here already, including the extension with a gas-pedal technique which is necessary to enable the timing dependability guarantees.

References

1. ARM: Cortex-a9 processor specifications. https://www.arm.com/products/processors/cortex-a/cortex-a9.php?tab=Specifications (2015)
2. Atmel: Sam4l xplained pro user guide. Atmel Corporation
3. M.A. Awan, S.M. Petters, Race-to-halt energy saving strategies. J. Syst. Archit. **60**(10), 796–815 (2014)

[2] Alternatively, core resources enabling more parallelism would also have enabled guarantees for a larger application workload or higher error rates.

4. Communication, T., Security: Spectrum monitoring and homeland security. https://www.thalesgroup.com/en/worldwide/security (2015)
5. S. Corbetta, W. Meeus, D. Rodopoulos, E. Cappe, F. Catthoor, A. Fritsch, System-wide reliability analysis on real processor and application under vdd and T stress, in *SELSE Silicon Errors in Logic System Effects* (2016)
6. S.V. Gheorghita, T. Basten, H. Corporaal, Intra-task scenario-aware voltage scheduling, in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (ACM, New York, 2005), pp. 177–184
7. S.V. Gheorghita, S. Stuijk, T. Basten, H. Corporaal, Automatic scenario detection for improved WCET estimation, in *Proceedings of the 42nd Annual Design Automation Conference* (ACM, New York, 2005), pp. 101–104
8. S.V. Gheorghita, T. Basten, H. Corporaal, Scenario selection and prediction for DVS-aware scheduling of multimedia applications. J. Signal Proces. Syst. **50**(2), 137–161 (2008)
9. C. Lee, M. Potkonjak, W.H. Mangione-Smith, MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, in *Proceedings of Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, 1997* (IEEE, Piscataway, 1997), pp. 330–335
10. NXP: i.mx 6dual/6quad automotive and infotainment applications processors data sheet, document no. imx6dqaec 3/2014. Tech. rep. (2014)
11. D. Rodopoulos, F. Catthoor, D. Soudris, Tackling performance variability due to RAS mechanisms with PID-controlled DVFS. IEEE Comput. Archit. Lett. **14**(2), 156–159 (2015)
12. N. Semiconductors, i.mx 6dual/6quad applications processor reference manual. http://www.nxp.com/assets/documents/data/en/reference-manuals/IMX6DQRM.pdf (2015). Rev 3
13. N. Zompakis, F. Catthoor, D. Soudris, Efficient configurations for dynamic applications in next generation mobile systems, ed. by A.D. Panagopoulos, in *Handbook of Research on Next Generation Mobile Communication Systems* (IGI Global, Oxford, 2015) Chap. 6, pp. 112–148

Chapter 5
DVAFS—Dynamic-Voltage-Accuracy-Frequency-Scaling Applied to Scalable Convolutional Neural Network Acceleration

Bert Moons and Marian Verhelst

Another extension to DVFS is DVAFS: dynamic-voltage-accuracy-frequency-scaling, introduced in [17, 20, 24]. Here, voltage and frequency are not modulated with different throughput requirements as is the case in DVFS, but in scenarios requiring different levels of accuracy. This goes clearly beyond the way system scenarios are exploited in the DVFS approaches of Chap. 4. This extension is however also complementary and usable concurrently with the earlier system scenario techniques. If a certain scenario only needs low internal computational precision, the supply voltages and frequency of a full digital system can be modulated, leading to significant energy savings compared to a high precision scenario. As such DVAFS can be situated in the field of approximate computing (AC) [4], a set of techniques that allow digital processing to approximate results and hence enable hardware to trade-off energy for accuracy.

Multiple application domains have proven to be tolerant to inaccurate or low-precision operations and to be able to gain from such a trade-off. Several works propose automated ways to estimate the impact of inaccurate computations on DSP applications. Either through using the statistics of the fixed-point quantization process [2, 25], through Monte Carlo methods, or through analytical modeling of error propagation [9]. A good overview of this field can be found in [1]. More specifically, Moons and Verhelst [17] show the DCT computations in a JPEG encoder can be performed at 4-bit accuracy with only 2 dB SNR-loss. In [5], various machine learning algorithms ranging from support vector machines to K-means clustering and specifically deep neural networks [15] are identified as fault-tolerant. Neural networks are used in different scenarios, some requiring different levels of internal computational precision. Depending on the complexity of the task of the network and on the specific neural network that is used [19, 21], the necessary

B. Moons · M. Verhelst (✉)
ESAT/MICAS, KU Leuven, Leuven, Belgium
e-mail: bert.moons@esat.kuleuven.be; marian.verhelst@esat.kuleuven.be

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6_5

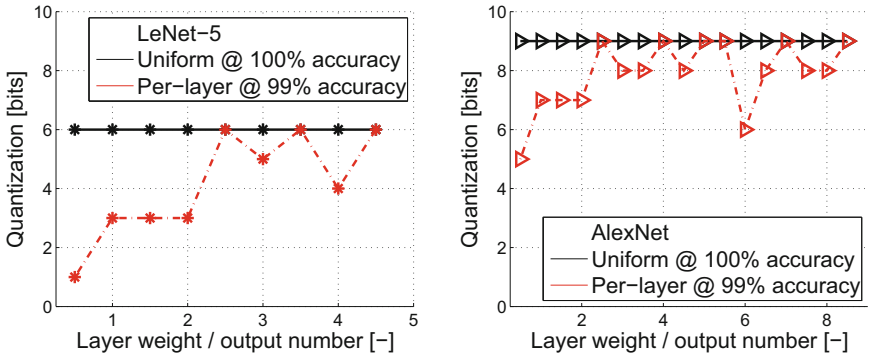


Fig. 5.1 Different tasks and networks require different internal computational precision while losing less than 1% relative accuracy [19]. The used precision can even vary from layer to layer. (**a**) 1 − 4b in LeNet-5 [13] on MNIST [14]. (**b**) 5-9b in AlexNet [10] on IMAGENET [23]

internal precision used in all filtering operations may vary. This is illustrated in Fig. 5.1. In this figure, the red line shows the number of bits necessary in every layer (fractional numbers are inputs/features, integer number are filter weights) if each layer is allowed a different computational precision and the network achieves 99% relative accuracy compared to a floating point baseline. The black lines show the number of bits if all features and weights are required to use the same number of bits. Figure 5.1a shows only 1 − 6b quantization is necessary in LeNet5 [13] operating on a simple handwritten digit recognition task (MNIST [14]). Figure 5.1b shows the necessary number of bits varies from 5-9b in AlexNet [10] operating on ImageNet [23]. Another example of such changing scenarios in visual face recognition is given in Sect. 3, where the necessary number of bits varies from 2 to 7b. The DVAFS technique can be used to minimize energy depending on the requirements of each of those scenarios.

The rest of this chapter discusses the basic concept of the DVAFS technique in Sect. 1, its performance on the block and system level in Sect. 2, and finally a silicon implementation and its typical scenarios in Sect. 3.

1 Exploiting Dynamic Precision Requirements in DVAFS

1.1 DAS: Dynamic-Accuracy-Scaling

An effective approach to dynamically scale the power consumption of digital arithmetic at constant throughput in function of the required computational precision is through truncation or rounding of a variable number of its input bits at run-time [24], see Fig. 5.2a. This introduces deviations in output quality caused by

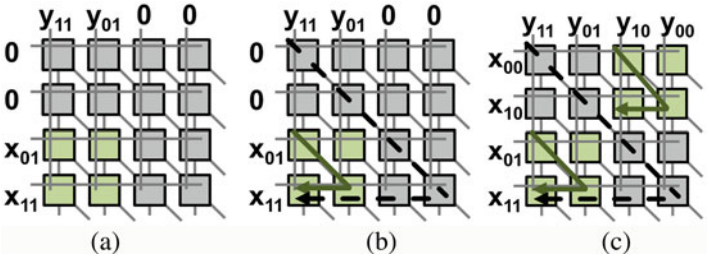


Fig. 5.2 A naive carry-save multiplier in multiple accuracy scaling modes. (**a**) DAS example. (**b**) DVAS example. (**c**) DVAFS example

Table 5.1 D(V)A(F)S parameters for the multiplier of Sect. 2.1

Parameter	4b	8b	12b	16b
k_0	12.5	3.5	1.4	1
k_1	12.5	3.5	1.4	1
k_2	1.2	1.1	1.02	1
k_3	3.2	1.82	1.45	1
k_4	1.53	1.27	1.02	1
N	4	2	1	1

higher quantization errors on the inputs and thus decreases computational accuracy. Simultaneously, it also reduces the circuit's internal switching activity and hence energy consumption. A good overview of these effects can be found in [1]. If leakage power is neglected, the power of such a dynamic-accuracy-scaling (DAS) system, P_{DAS}, is split into an accuracy-scalable part (as), of which the activity is modulated with scaled precision and a non-accuracy-scalable (nas) part which does not modulate with scaled precision:

$$P_{DAS} = \frac{\alpha_{as}}{k_0} C_{as} f V^2 + \alpha_{nas} C_{nas} f V^2 \tag{5.1}$$

where α is the circuit's switching activity, f is the operating frequency, C is the technology dependent switching capacitance, and V is the supply voltage. k_0 is a precision, circuit, and architecture dependent scaling parameter. An example of k_0 for a real multiplier is given in Table 5.1.

1.2 DVAS: Dynamic-Voltage-Accuracy-Scaling

A DAS data path element, capable of modulating the computational precision of its input, will exhibit a different critical path length in function of the used computational precision. This is illustrated in Fig. 5.2b for a simplified $1 - 4$b multiplier with two operating modes (2b and 4b). In the high precision mode example of Fig. 5.2b all building blocks will be active, resulting in a high switching

activity and a long critical path. However, in the low-precision case, only the most significant bits (MSB) are used, resulting in a lower switching activity and shorter critical path. The DVAS concept [17, 18] exploits this critical path scaling by dynamically scaling voltage in function of the used number of bits, without inducing timing errors. At a fixed frequency, this positive timing slack can then be transformed into energy savings through a lower supply voltage.

These combined effects—(1) reduced switching activity and (2) shorter critical paths allowing lower supply voltages—have a major impact on the system's dynamic power consumption. The power P_{DVAS} of a DVAS-system is split into a voltage-accuracy scalable vas, a non-voltage-accuracy scalable $nvas$, and a non-accuracy-scalable nas part, given in Eq. (5.2).

$$P_{DVAS} = \frac{\alpha_{vas}}{k_1} C_{vas} f \left(\frac{V_{vas}}{k_2} \right)^2 + \frac{\alpha_{nvas}}{k_x} C_{nvas} f (V_{nvas})^2 + \alpha_{nas} C_{nas} f V_{nas}^2 \tag{5.2}$$

The critical path only scales in arithmetic building blocks, such as multipliers and adders, which are considered vas. In $nvas$ blocks such as memories, register files, and multiplexers, only the switching activity drops at lower precision. Finally, a considerable portion of the original power consumption in nas blocks, such as control blocks and decoders, does not scale under reduced precision. As the vas, $nvas$, and nas parts operate at different voltages, the design should be split into separate power domains.

1.3 DVAFS: Dynamic-Voltage-Accuracy-Frequency-Scaling

Subword-parallel DVAFS [22] improves further upon the energy savings of DVAS by reusing inactive arithmetic cells at reduced precision. The simplified example in Fig. 5.2c shows a multiplier that can process two subword operations per cycle if precision is scaled to 2b or lower. If computational throughput is kept constant, this allows to drop the full system's frequency and hence its voltage significantly below DVAS values. As a result, DVAFS is the first dynamic AC technique which simultaneously lowers all run-time adaptable parameters influencing power consumption: activity α, frequency f, and voltage V. In contrast to DVAS, which can only save energy in precision-scaled arithmetic blocks, DVAFS allows lowering f and V of the full system, including control units and memory, hereby shrinking nas energy overheads drastically at low precision.

These combined effects—(1) reduction of switching activity, (2) shorter critical paths allowing lower supply voltages, (3) subword-parallel operation allowing lower operating frequencies at constant computational throughput—have a major impact on the system's power consumption. The dynamic power P_{DVAFS} of a DVAFS-system is given in Eq. (5.3)

$$P_{DVAFS} = \frac{\alpha_{vas}}{k_3} C_{vas} \frac{f}{N} \left(\frac{V_{vas}}{k_4} \right)^2 + \frac{\alpha_{nvas}}{k_z} C_{nvas} \frac{f}{N} \left(\frac{V_{nvas}}{k_y} \right)^2 + \alpha_{nas} C_{nas} \frac{f}{N} \left(\frac{V_{nas}}{k_5} \right)^2 \quad (5.3)$$

where N is the subword parallelism, processing $(16/N)b \times N$ words per cycle.

2 DVAFS Performance Analysis

2.1 Performance of a DVAFS Multiplier

The performance and energy-accuracy trade-off of a DVAFS-equipped 16b Booth-encoded Wallace-tree multiplier [20] is assessed through detailed simulations in a 40 nm LP LVT technology with a nominal supply voltage of 1.1 V. The multiplier is synthesized with commercially available cells in a standard digital flow, using a multi-mode optimization, ensuring the critical path indeed decreases when less bits are used. Conservative wire models are used for synthesis and power estimations. In these simulations, multiplier throughput is kept constant at $T = 1$ words/cycle $\times$ 500 MHz $= 2$ words/cycle $\times$ 250 MHz $= 4$ words/cycle $\times$ 125 MHz $= 500$ MOPS, as shown in Fig. 5.3a. However, the word size varies accordingly from 16b down to 4b. Figure 5.3b shows the effect of D(V)A(F)S on the positive circuit delay slack. Without voltage scaling positive slack increases up to 1 ns if 4b words are processed in the D(V)AS case and up to 7 ns due to the 125 MHz clock in the 4×4b D(V)AFS case. Figure 5.3c shows this slack can be compensated for through lowering the supply voltage at constant throughput. For DVAS, a reduction down to 0.9 V is achievable. In DVAFS, the voltage can go down to 0.75 V, or an additional decrease in energy of 55%. Figure 5.3a and d show the respective operating frequencies and activity reduction of the multiplier computing at different levels of accuracy. Switching activity drops 12.5$\times$ in the DVAS-case and 3.2$\times$ in the DVAFS-case at 4b. However, in DVAFS the frequency is lowered to 125 MHz, while it remains

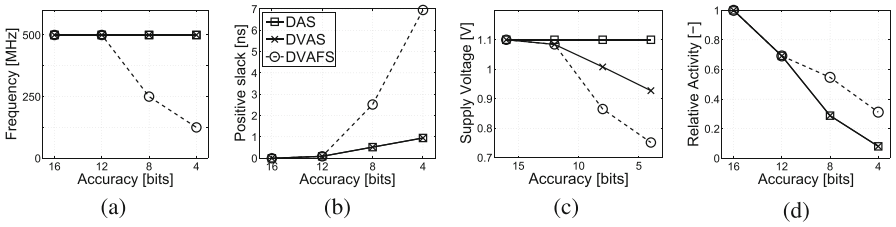


Fig. 5.3 Frequency, slack, voltage, and activity in a custom Booth-encoded Wallace-tree subword-parallel DVAFS multiplier in DAS-, DVAS-, and DVAFS-modes. (**a**) Frequency. (**b**) Slack @ 1.1 V. (**c**) Voltage at zero slack. (**d**) α

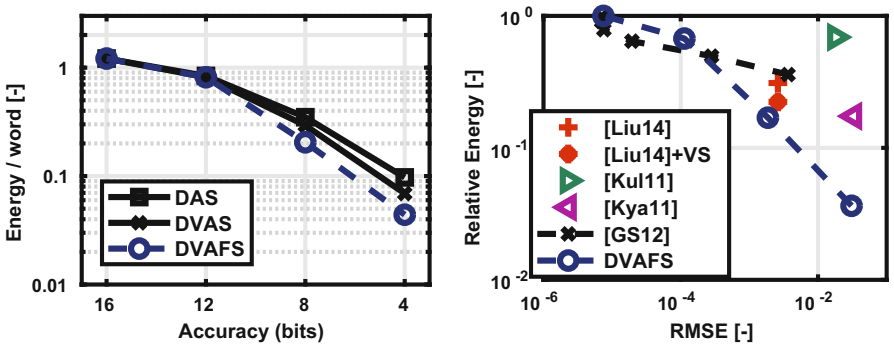


Fig. 5.4 Comparison of energy gains in a custom Booth-encoded Wallace-tree DVAFS multiplier to [6, 11, 12, 16] as a function of the used precision. (**a**) D(V)A(F)S. (**b**) DVAFS vs [6, 11, 12, 16]

constant at 500 MHz for all accuracies in DVAS. This analysis allows extracting parameters k_i and N for this multiplier, as given in Table 5.1.

The combination of these reductions in energy consumption leads to global energy-accuracy curves shown in Fig. 5.4a. Here, the total energy consumption per computed word of the multiplier (normalized to a non-reconfigurable 16b multiplier) is compared. Note that processing at 16b in the DVAFS multiplier comes at a slight energy penalty, as this reconfigurability leads to a 21% overhead at full precision. At full 16b resolution the reconfigurable multiplier consumes 2.63 pJ/word compared to the 2.16 pJ/word baseline in this technology. The DAS case illustrates the sole effect of the decrease in α due to accuracy scaling. Energy drops more significantly in DVAS due to the added voltage scaling. DVAFS achieves energy savings of more than 95% of the baseline at 4×4 DVAFS processing.

The energy-accuracy trade-off in DVAFS is superior to other AC techniques, both in terms of energy reduction as well as in terms of energy-accuracy dynamic range. Figure 5.4b compares the curve of the DVAFS multiplier with [6, 11, 12, 16], with energy relative to the respective fully accurate implementation in each reference and accuracy expressed in terms of root-mean-square-error (RMSE). Although [6] consumes less energy at high accuracy, the energy consumption is higher at accuracies lower than 1e − 4 RMSE. Liu et al. [16], Kulkarni et al. [11], and Kyaw et al. [12] do not allow a run-time adaptable trade-off and consume more energy per word at the same level of accuracy.

2.2 Performance of a DVAFS SIMD Processor

To show the advantages of DVAFS at the system level, a DVAFS-compatible SIMD RISC vector processor is simulated and implemented in an application-specific instruction set processor (ASIP) design tool [26], using the same 40 nm LP LVT

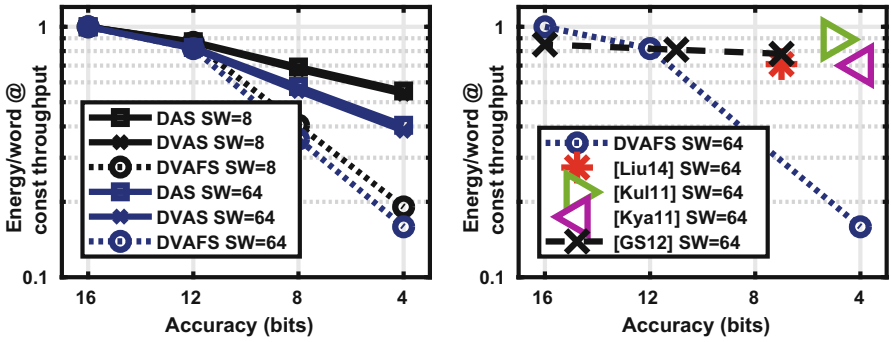


Fig. 5.5 Comparison of energy gains in a DVAFS SIMD processors to [6, 11, 12, 16]. (**a**) DVAFS. (**b**) DVAFS vs [6, 11, 12, 16]

technology as in Sect. 2.1. The processor has a parameterized SIMD width SW, denoting the number of data path units and memory banks. Every SIMD-unit can scale its precision across $1 \times 1 - 16$b, $2 \times 1 - 8$b, $4 \times 1 - 4$b DVAFS modes.

In order to support DVAFS, the processor contains all necessary rounding circuitry. All memories are in a separate power domain (V_{MEM}), at a fixed 1.1 V to maintain reliable operation. All other parts of the processor are in two power domains with variable supply voltage (V_{as} and V_{nas}). As a benchmark, a large convolution kernel is run on the SIMD processor.

The performance of the SIMD processor is illustrated in Fig. 5.5, where the energy of the complete processor per processed word is plotted against the used computational accuracy at constant throughput for processor instantiations with different SIMD widths SW. The $SW-$processor in the 1×16b mode at 500 MHZ is used as a baseline. The maximal decrease in energy consumption is achieved at the 4×4b DVAFS mode, with a reduction of 85% compared to the baseline. Gains are more modest in the DAS and DVAS (60%) case as the contribution of the as data path to the full processor's energy consumption is smaller. The highly parallel $SW = 64$ processor has a larger percentage of its energy consumption in the arithmetic circuits and thus has a larger potential for energy reduction than its $SW = 8$ counterpart. In a sole DVAS-system, the parallelism should be large for significant energy gains. In DVAFS, energy scales down significantly also at low levels of parallelism (low SW), as the supply voltage is also lowered in the non-accuracy-scalable (nas) parts. Figure 5.5b also compares the DVAFS techniques with the references from [11, 12, 16], through extrapolating their block-level performance to the simulated SIMD processor. As DVAFS allows system-wide gains, not only in the computational units, it outperforms all of them in the energy-accuracy space.

DVAFS-compatible silicon prototypes, with voltage- and critical path scaling have been developed, as is discussed in Sect. 3.2.

3 A DVAFS Prototype

Typically, inferring state-of-the-art neural networks comes at very high energy cost, too high for embedded applications. However, because of the fault tolerance of these algorithms, they can be executed using low or varying computational precision, depending on the scenario they are used in. The DVAFS-compatible Envision chip [22], introduced here, allows reducing energy consumption accordingly, see Sect. 3.1. It can be used in a hierarchical face recognition cascade, see Sect. 3.2.

3.1 Envision: A DVAFS-Compatible CNN Processor

The Envision chip, implemented in a 28 nm FDSOI technology and illustrated in Fig. 5.6 is a C-programmable processor, both exploiting algorithmic sparsity [18, 19] and the dynamic accuracy scaling discussed here. The chip contains around 1.95 M gates on an area of 1.87 mm^2, split into three power domains (as indicated in Fig. 5.6b) to support granular DVAFS. Figure 5.6a shows the chip contains a 2D-SIMD array of 256 multiply accumulate units (MAC) for convolutional kernels, vector and scalar units, a flexible memory architecture, and a DMA with IO en/decoding. As discussed in Sect. 1.3, a DVAFS processor should support subword-parallel $N \times 16/N$bits operating modes, with N the number of parallel subwords. Hence, at 200 MHz, Envision's 256 processing units achieve a peak 102 GOPS in the $1 \times 1 - 16$b mode and up to 408 GOPS in the $4 \times 1 - 4$b mode. The chip has 132 kB on-chip data memory and 16 kB on-chip program memory for its 16b instruction set.

The energy efficiency of Envision is illustrated in Fig. 5.7, both at constant 200 MHZ frequency and at a 76 GOPS constant throughput. These measurements

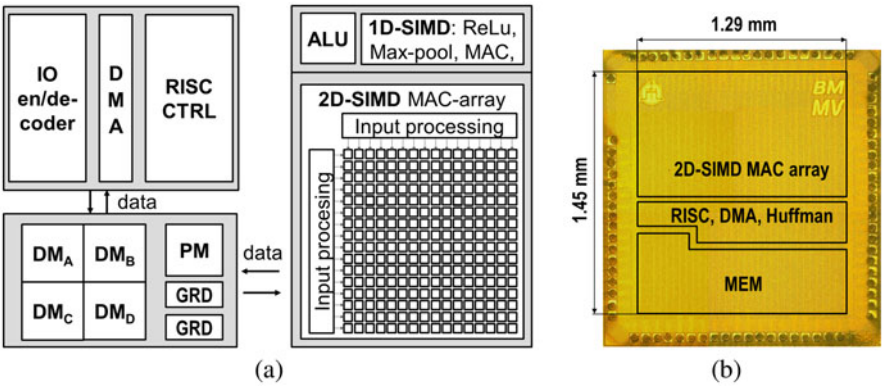
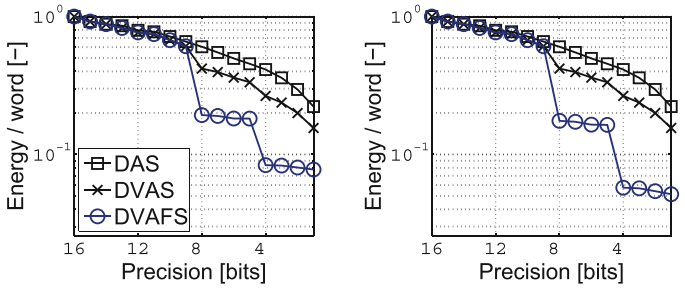


Fig. 5.6 (**a**) High-level specifications of Envision. (**b**) Chip photograph of Envision. *N* is the level of subword parallelism



(a) Constant f=200MHz (b) Constant T=76GOPS

Fig. 5.7 (**a**) Relative energy consumption of Envision at 200 MHz. These numbers are normalized to the maximal power consumption of 300 mW at 16b. (**b**) Energy consumption per word at 76 GOPS. At constant throughput, frequency and supply voltage can be lowered further at lower accuracy leading to higher efficiency gains

are done on a 5×5 CONV layer with a typical MAC-efficiency of 73% or $0.73 \times 256 \times 2 \times f$ operations per second. In DAS and DVAS, the energy gains at reduced computational accuracy are limited, as the non-scalable parts of the processor such as the instruction decoder and memories become dominant. Figure 5.7a shows the processor consumes 2.4× and 3.8× less energy per 4b operation than for 16b full precision, in the DAS and DVAS cases, respectively.

If frequency is kept constant at 200 MHz, the chip consumes 300 mW at full 16b precision and 76 real GOPS, leading to an efficiency of 250 GOPS/W. At the nominal frequency of 200 MHz and in the $N = 4$, 4×4b mode, the chip consumes 104 mW and achieves $4 \times 76 = 304$ real GOPS or 2.8 TOPS/W.

Efficiency can be further improved at low computational accuracy through exploiting the frequency-scaling capabilities of DVAFS. If frequency is scaled under constant computational throughput, the supply voltage of the whole system can be lowered further, leading to higher efficiency gains. This is illustrated in Fig. 5.7b, where in the 4×4b full DVAFS mode, frequency is scaled down from 200 MHz to 50 MHz to maintain the 76GOPS throughput. In this case, power consumption scales from 300 mW down to 18 mW or 4.2 TOPS/W, or a 6.9× and 4.1× improvement over DAS and DVAS, respectively.

As indicated above, different scenarios require different levels of computational precision. A simple handwritten digit classification task can be performed using $1-$ 6b fixed-point precision, while more complex classification on ImageNet [23] and LFW [8] requires 4-9b operation. The DVAFS-based CNN processor thus allows performing simple tasks at a much higher energy efficiency compared to more complex tasks. This is illustrated in Table 5.2 showing the energy efficiency of the different convolutional layers of VGG16 (3.3 fps), AlexNet (47 fps), and LeNet-5 (13 kfps) on Envision. Envision achieves 1.8 TOPS/W on AlexNet, significantly outperforming non-scalable [3] (0.16 TOPS/W) and DVAS-only [18] (0.94 TOPS/W) implementations. Switching between these operating modes can be performed in software from CNN layer to layer by changing a single configuration register. In

Table 5.2 Power consumption in VGG16, AlexNet, and LeNet-5 on Envision [22] due to sparsity [18] and DVAFS

Layer	Mode	f [MHz]	V [V]	Wght. [b]	In. [b]	Wght. sp.[%]	In. sp. [%]	MMACS/ frame	Power [mW]	Eff. [TOP-S/W]
VGG1	2 × 8b	100	0.80	5	4	5	10	87	25	2.1
VGG2-13	2 × 8b	100	0.80	5	6	25–75	30–82	462–1850	19–35	1.5–2.8
Total	–	–	–	–	–	–	–	15,346	26	2
AlexNet1	2 × 8b	100	0.80	7	4	21	29	104	37	2.7
AlexNet2	2 × 8b	100	0.80	7	7	19	89	224	20	3.8
AlexNet3	1 × 16b	200	1.03	8	9	11	82	150	52	1
AlexNet4-5	1 × 16b	200	1.03	9	8	4	72	112	58–62	0.8–0.9
Total	–	–	–	–	–	–	–	666	44	1.8
LeNet1	4 × 4b	50	0.65	3	1	35	87	0.3	5.6	13.6
LeNet2	2 × 8b	100	0.80	4	6	26	55	1.6	29	2.6
Total	–	–	–	–	–	–	–	1.9	25	3

the context of neural networks, the complexity and control of switching between different scenarios is overall very limited. In different, less regular applications, with larger code-bases and loop nests, the scenario-control and DVAFS-mode switches should be implemented in a fully systematic and automated way.

3.2 Envision in a Face Recognition Hierarchy

Hierarchical cascaded processing [7] is a multi-stage generalization of wake-up based detection and recognition. The technique outperforms classical wake-up schemes, as a cascaded hierarchy optimally exploits the statistics of the input data.

An example of such a hierarchy in the context of face recognition is shown in Fig. 5.8. Rather than using very complex and costly face recognizers in an always-on manner, it is more efficient to build the same functionality out of a cascade of cheaper, but increasingly complex scenarios. A first stage could be a very cheap face detector that filters out backgrounds. A second stage can be an owner-detector, as a detected face is most likely the device's owner in the context of mobile phones. If the detected face is not the owner, a third stage can classify ten most-likely people before powering on the next stage. On average, this scenario-based adaptivity strategy can save 4 orders of magnitude compared to a setup with only 1 complex stage and 2 orders of magnitude compared to the classic wake-up scenario at the same performance [7].

In such a hierarchy, not only the model size of the neural network and its number of necessary computations vary from stage to stage, but also the number of bits used. A simple task in an early stage typically requires less bits than a more complex task in a later stage. The Envision processor is able to exploit this by dynamically tuning its internal computational precision, hereby minimizing energy consumption

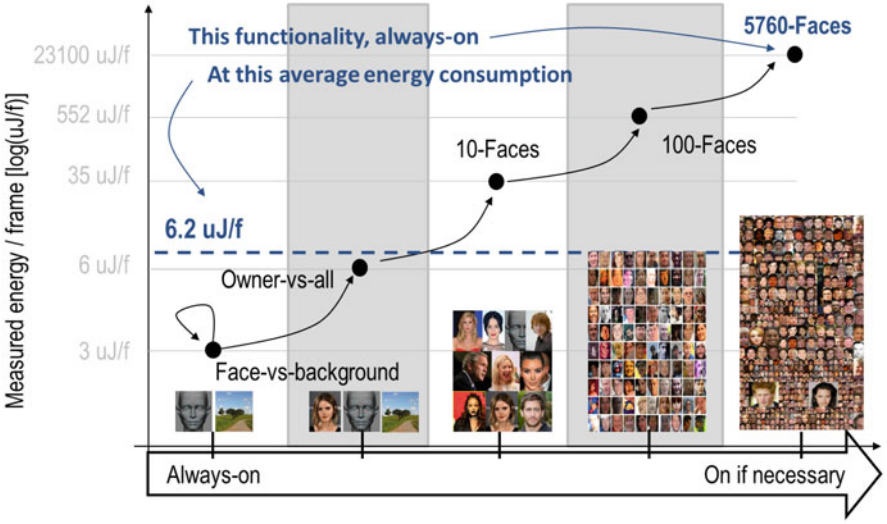


Fig. 5.8 A 5-stage hierarchical face recognition example. The average energy/frame estimate is when stages 1–5 are used on [100, 1,0.5, 0.1,0.01]% of the frames

Table 5.3 Different scenarios require different levels of computational precision for accurate results

Task	Input	# Classes	Feature [bits]	Weight [bits]	CONV size[kb]	CONV ops [MMACs]	Eff. [TOPS/W]	En./ Frame[J]	Acc. [%]
Face vs background	32 RGB	2	2–3	2–4	22	6	4.2	3u	94.3
Owner vs all	32 RGB	3	3	3–4	42	12	4	6u	95.9
10 faces	32 RGB	12	4–6	4	112	30	1.8	35u	95.4
100 faces	224 RGB	104	4–6	5–6	742	500	1.8	552u	94
>5000 faces	224 RGB	5760	5–6	6–7	15k	15.4k	1.3	23.2m	95

Envision can trade this in for energy efficiency

in any stage of such a hierarchy. Table 5.3 shows the specifications of the different stages in the hierarchy of Fig. 5.8. The first wake-up stage is very small and requires only 2-4b of precision in order to get to a reliable result. Envision can then run these small networks at an average efficiency of more than 4 tera-operations per second per Watt (TOPS/W), due to the advanced precision, voltage, and frequency scalability of DVAFS. The last stage is more complex and requires 5-6b operators to achieve a high accuracy. Because of this, energy efficiency drops by almost 3× at the same frequency to 1.3 TOPS/W. Each of the separate scenario's in the face recognition hierarchy can hence be individually optimized through DVAFS. Using the hierarchical approach in Fig. 5.8, when stages 1–5 are used on [100, 1 ,0.5, 0.1,

0.01]% of the frames, the average energy consumption per mini-frame is brought down from 23 mJ/frame to 6.2 μJ/frame at the same accuracy. Without DVAFS, if all scenarios would run at an efficiency of 1.3 TOPS/W using 5-6b operators, this average number would go up to at least 13 μJ/frame. Hence, here DVAFS leads to more than a 2× advantage over an optimized hierarchy running at fixed 5-6b precision. A fixed-point 16b non-DVAFS hierarchical solution would be 4× more expensive.

4 DVAFS Overview

Dynamic-voltage-accuracy-frequency-scaling (DVAFS) is a generalization of DVFS, in which voltage and frequency are modulated with the varying accuracy requirements of a scenario rather than solely with its varying throughput requirements. Hence, if a low internal computational precision is allowed, a digital system can reduce its energy consumption dynamically through DVAFS. Typical examples of applications where this is useful are in object recognition using neural networks. Here simple applications, such as digit recognition, only need low-precision computations, while more complex applications, such as 100-class object recognition, require higher precision. The DVAFS technique outperforms other similar techniques in the dynamic energy-vs-accuracy space. It consumes 3× less energy on the block-level and up to 5× less energy on the system level compared to the state-of-the-art. Envision, a silicon-implemented DVAFS-compatible CNN processor, can scale its energy efficiency from 0.3 TOPW/S at 16b precision up to 4 TOPS/W at 4b through DVAFS and up to 10 TOPS/W for sparse 4b data flows. In a hierarchical face recognition example, Envision's DVAFS capabilities offer a 2× advantage over a fixed low-precision implementation and an 8× advantage over a fixed 16b implementation.

References

1. B. Barrois, Methods to evaluate accuracy-energy trade-off in operator-level approximate computing, Ph.D. thesis, Universite de Rennes, 2017
2. B. Barrois, K. Parashar, O. Sentieys, Leveraging power spectral density for scalable system-level accuracy evaluation, in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016* (IEEE, Piscataway, 2016), pp. 750–755
3. Y.-H. Chen, T. Krishna, J. Emer, V. Sze, Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. *ISSCC Digest of Technical Papers*, pp. 262–263, 2016
4. S. Chippa, V. Venkataramani, S.T. Chakradhar, K. Roy, A. Raghunathan, Approximate computing: an integrated hardware approach, in *Asilomar Conference on Signals, systems and computers*, 2013
5. V.K. Chippa, S.T. Chakradhar, K. Roy, A. Raghunathan, Analysis and characterization of inherent application resilience for approximate computing, in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013

6. M. de la Guia Solaz, W. Han, R. Conway, A flexible low power DSP with a programmable truncated multiplier, in *TCAS-I*, 2012
7. K. Goetschalckx, B. Moons, S. Lauwereins, M. Andraud, M. Verhelst, Optimized hierarchical cascaded processing. IEEE J. Emerging Sel. Top. Circuits Syst. **8**(4), 884–894 (2018)
8. G.B. Huang, M. Ramesh, T. Berg, E. Learned-Miller, Labeled faces in the wild: a database for studying face recognition in unconstrained environments. Technical report, Technical Report 07-49, University of Massachusetts, Amherst, 2007
9. J. Huang, J. Lach, G. Robins, A methodology for energy-quality tradeoff using imprecise hardware, in *Proceedings of the 49th Annual Design Automation Conference* (ACM, New York, 2012), pp. 504–509
10. A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in *Advances in Neural Information Processing Systems*, ed. by F. Pereira, C.J.C. Burges, L. Bottou, K.Q. Weinberger, vol. 25, pp. 1097–1105 (Curran Associates, Inc., Red Hook, 2012)
11. P. Kulkarni, P. Gupta, M. Ercegovac, Trading accuracy for power with an underdesigned multiplier architecture, in *International Conference on VLSI Design*, 2011
12. K.Y. Kyaw et al., Low-power high-speed multiplier for error-tolerant application, in *Electron Devices and Solid-State Circuits (EDSSC)*, 2011
13. Y. LeCun, C. Cortes, The MNIST database of handwritten digits, 1998
14. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
15. Y. LeCun, Y. Bengio, G. Hinton, Deep learning. Nature **521**(7553), 436–444 (2015)
16. C. Liu, J. Han, F. Lombardi, A low-power, high performance approximate multiplier with configurable partial error recovery, in *Design, Automation and Test in Europe (DATE)*, 2014
17. B. Moons, M. Verhelst, DVAS: dynamic voltage accuracy scaling for increased energy-efficiency in approximate computing, in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2015
18. B. Moons, M. Verhelst, A 0.3-2.6 tops/w precision-scalable processor for real-time large-scale ConvNets, in *Proceedings of the IEEE Symposium on VLSI Circuits*, pp. 178–179, July 2016
19. B. Moons, B. De Brabandere, L. Van Gool, M. Verhelst, Energy-efficient convnets through approximate computing, in *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 1–8, 2016
20. B. Moons, R. Uytterhoeven, W. Dehaene, M. Verhelst, DVAFS: trading computational accuracy for energy through dynamic-voltage-accuracy-frequency-scaling, in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE, Piscataway, 2017), pp. 488–493
21. B. Moons, K. Goetschalckx, N. Van Berckelaer, M. Verhelst, Minimum energy quantized neural networks, in *Asilomar Conference on Signals, Systems and Computers*, 2017
22. B. Moons, R. Uytterhoeven, M. Dehaene, W. Verhelst, Envision: a 0.26-to-10 tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI, in *International Solid-State Circuits Conference (ISSCC)*, 2017
23. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein et al., ImageNet large scale visual recognition challenge. Int. J. Comput. Vis. **115**(3), 211–252 (2015)
24. S. Venkataramani et al., Quality programmable vector processors for approximate computing, in *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013
25. B. Widrow, Statistical analysis of amplitude-quantized sampled-data systems. Trans. Am. Inst. Electr. Eng. II: Appl. Ind. **79**(6), 555–568 (1961)
26. B. Wu, M. Willems, Rapid architectural exploration in designing application-specific processors, in *ASIP Designer Whitepaper*, 2015

Chapter 6
Run-Time Exploitation of Application Dynamism for Energy-Efficient Exascale Computing

Per Gunnar Kjeldsberg, Robert Schöne, Michael Gerndt, Lubomir Riha, Venkatesh Kannan, Kai Diethelm, Marie-Christine Sawley, Jan Zapletal, Andreas Gocht, Nico Reissmann, Ondrej Vysocky, Madhura Kumaraswamy, and Wolfgang E. Nagel

1 Introduction and Context

In the embedded systems domain, energy efficiency has been a main design constraint for more than two decades. More recently, this has also become a major concern in high performance computing (HPC). Even though these two domains

P. G. Kjeldsberg (✉) · N. Reissmann
Norwegian University of Science and Technology, NTNU, Trondheim, Norway
e-mail: pgk@ntnu.no; nico.reissmann@ntnu.no

R. Schöne · A. Gocht · W. E. Nagel
Technische Universität Dresden, Dresden, Germany
e-mail: robert.schoene@tu-dresden.de; andreas.gocht@tu-dresden.de; wolfgang.nagel@tu-dresden.de

M. Gerndt · M. Kumaraswamy
Technische Universität München, München, Germany
e-mail: gerndt@in.tum.de; kumarasw@in.tum.de

L. Riha · J. Zapletal · O. Vysocky
VSB - Technical University of Ostrava, Ostrava, Czech Republic
e-mail: lubomir.riha@vsb.cz; jan.zapletal@vsb.cz; ondrej.vysocky@vsb.cz

V. Kannan
Irish Center for High-End Computing, Galway, Ireland
e-mail: venkatesh.kannan@ichec.ie

K. Diethelm
Gesellschaft für numerische Simulation, Braunschweig, Germany

University of Applied Sciences Würzburg-Schweinfurt, Schweinfurt, Germany
e-mail: diethelm@gns-mbh.com

M.-C. Sawley
Intel ExaScale Labs, Paris, France
e-mail: marie-christine.sawley@intel.com

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6_6

are different in many respects, techniques that have proven to be efficient in one may also be beneficial in the other. The split design-time and run-time approach of system-scenario-based design, which is described in this book, is being applied successfully in embedded systems, both to increase performance and to reduce power and energy consumption [7, 8, 12]. In the HPC domain, auto-tuning is used either at design-time to statically tune the system configuration or at run-time through compute intensive estimation of the dynamic requirements [2, 15]. Combining these approaches from embedded systems and HPC has the potential of giving substantial synergies in both domains.

A constantly growing demand for data center computing performance leads to the installation of increasingly powerful and complex systems, characterized by a rising number of CPU cores as well as increasing heterogeneity. This makes optimization of HPC applications a complex task, which demands significant programming effort and high levels of expertise. With a growing computational performance, there is typically also an increase in a system's energy consumption, which in turn is a major driver for the total cost of ownership of HPC systems. Furthermore, limitations to chip temperature and cooling capabilities can make the performance of Exascale HPC systems power-bound. However, developers commonly focus on the implementation and improvement of algorithms with regard to accuracy and performance, neglecting possible improvements to energy efficiency. Often, programmers lack the platform and hardware knowledge required to exploit these measures, which is an important obstacle for their use both in the embedded and HPC domains.

The European Union Horizon 2020 project READEX [6] (Run-time Exploitation of Application Dynamism for Energy-efficient eXascale computing) tackles these challenges by embracing the significant potential for improvements to performance and energy efficiency that result from dynamic resource requirements of HPC applications similar to those seen in embedded systems. Examples are alternating application regions and load-changes at application run-time. Such dynamism can be found in current HPC applications, including weather forecasting, molecular dynamics, or adaptive mesh-refinement applications.

These applications often operate in an iterative manner, e.g., using a time step loop as the main control flow. Each iteration of such a program loop can be regarded as a phase of the application execution. In this context, intra-phase dynamism describes the changes in resource requirements and computational characteristics between different code regions executed by a single iteration, e.g., the change between memory- and compute-bound kernels. Intra-phase dynamism can be exploited by adjusting the system to the resource requirements of the current code region. Alternatively, inter-phase dynamism describes the changes in application behavior between iterations or phases. As the execution progresses, the required computation can vary, either on single processes—causing imbalances—or on all processes with a homogeneous rise in computational complexity on all processing elements.

As discussed in Chap. 1, it is expected that applications running on future embedded systems platforms will exhibit even higher levels of dynamism. This will

be mainly due to the increased demand for data movement performance between processing elements, both on intra- and inter-node levels, and more complex multi-level memory hierarchies. Furthermore, the rise of many-core co-processors and accelerators introduces new degrees of freedom such as offloading and scheduling. For extreme-scale HPC systems, this trend will be even more critical.

The READEX project has developed and implemented a tools-aided methodology that enables HPC application developers to exploit dynamic application behavior when run on current and future extreme parallel and heterogeneous multi-processor platforms. READEX combines and extends state-of-the-art technologies in performance and energy efficiency tuning for HPC with dynamic energy optimization techniques for embedded systems. Many of the techniques developed for HPC systems can also be fed back to the embedded systems domain, in particular with the increasing performance seen in embedded multi-processor system-on-chip [10].

The general concept of the READEX project is to handle application energy efficiency and performance tuning by taking the complete application life-cycle approach. This is in contrast to other HPC approaches that regard performance and energy tuning as a static activity, which takes place in the application development phase. With inspiration from system-scenario-based design, READEX has developed a (semi-)automatic dynamic tuning methodology spanning the development (design-time) and production/maintenance (run-time) phases of the application life-cycle. Furthermore, a novel programming paradigm for application dynamism was developed that enables domain experts to pinpoint parts of the application and/or external events that influence the dynamic behavior. This can reduce the energy consumption even further, compared to a purely automatic approach.

The rest of this chapter is organized as follows: After a review of related work and project background in Sect. 2, a description of the READEX concepts is given in Sect. 3. This is followed by results from experiments with a realistic industrial HPC application in Sect. 4. Section 5 concludes this chapter with a summary.

2 Auto-tuning of HPC Systems

Given that system-scenario-based design is fully covered in other chapters of this book, this section focuses on how dynamic behavior is handled in HPC systems.

While a small number of dynamic auto-tuning methodologies and tools existed for run-time optimizations [3, 20], there is currently no single standalone dynamic auto-tuning framework with the capability to target the full breadth of large-scale HPC applications being used in academia and industry, both now and on the road to Exascale.

Still, several EU research projects are approaching the challenge of tuning for performance and energy efficiency by either introducing entirely new programming models or leveraging existing prototype languages. An example of the latter is the ENabling technologies for a programmable many-CORE (ENCORE) project [5],

which aims to achieve massive parallelism relying on tasks and efficient task scheduling using the OmpSs programming model [19]. The READEX project takes a different approach by developing a new generic programming paradigm, which allows to express and to utilize dynamism of applications in the automatic tuning process.

The Performance Portability and Programmability for Heterogeneous Many-core Architectures PEPPHER project [1] has developed a methodology and framework for programming and optimizing applications for single-node heterogeneous many-core processors to ensure performance portability. With Intel as a key partner in the project, READEX goes one step further and provides a framework that supports the heterogeneity of the system in the form of tuning parameters, which enable large-scale heterogeneous applications to dynamically (and automatically) adapt heterogeneous resources according to run-time requirements.

The ANTAREX project [18] creates a domain specific language (DSL), which distributes the code between multi-core CPUs and accelerators. An extra compilation step is introduced to translate the DSL into the intended programming language. While our work targets conventional HPC clusters, the ANTAREX project focuses on ARM-based systems.

Nowadays, most performance engineering tools focus on collecting and presenting information to the users, while only a few focus on the automation of the performance optimization process (auto-tuning). One example of the latter is the periscope tuning framework (PTF) developed in the EU FP7 ICT AutoTune project [2, 15]. PTF automatically finds optimized system configurations for whole application runs, effectively averaging the benefits of system adaptation over the whole run-time of the application (static tuning). With these static auto-tuning techniques, improvements in energy efficiency of up to 10% for application runs have been achieved while keeping the performance degradation to a few percent [4].

PTF's main principles are the use of formalized expert knowledge and strategies, an extensible and modular architecture based on tuning plugins, automatic execution of experiments, and distributed, scalable processing. PTF provides a number of predefined tuning plugins, including:

- Dynamic voltage and frequency scaling (DVFS)
- Compiler flags selection
- MPI run-time environment settings
- Parallelism capping in OpenMP
- MPI master–worker pattern settings

PTF also provides an interface for the development of new plugins. It builds on the common performance measurement tools infrastructure Score-P [11], which has proven to be scalable on current petascale systems.

3 The READEX Concept

The READEX concept combines the system scenarios methodology with automatic energy and performance tuning into a holistic tools-aided methodology, spanning major parts of the HPC application life-cycle, i.e., application development (design-time) and production runs (run-time). Figure 6.1 provides a high-level overview of the methodology.

3.1 Application Instrumentation and Analysis Preparation

During the first step of the methodology, the application is instrumented by inserting probe functions around different regions in the application code. A region can be any arbitrary part of the code, for instance, a function or a loop nest. The instrumentation itself can be done automatically or by letting the user provide application domain knowledge using a new programming paradigm.

The programming paradigm enables users to expose parameters, which describe the dynamic behavior of the application to the READEX tool suite. These applica-

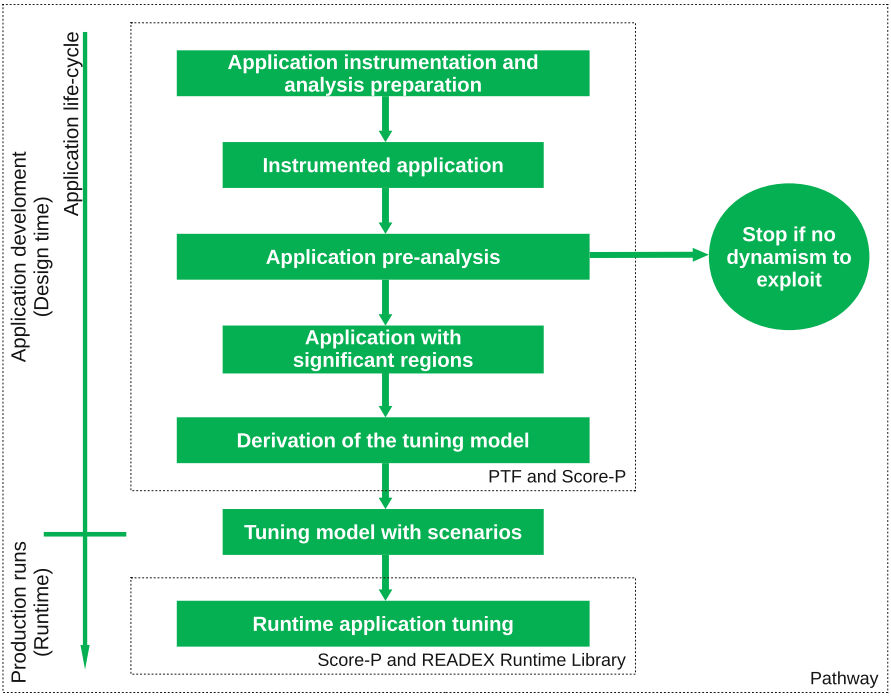


Fig. 6.1 Overview of the READEX methodology

tion domain parameters enhance the identification of different system scenarios and hence extend the adaptation of system configurations in order to improve overall application performance and energy characteristics.

One example of these parameters is the definition of different input data sets. Exposing such information to READEX enables the tools to attribute different compute characteristics to the varying input. The paradigm also enables developers to expose additional application-level tuning parameters to the tuning process, e.g., alternative code-paths that will be chosen based on the provided identifiers.

In addition to the optional information provided by the user, probe functions are automatically inserted around user code regions and by linking instrumented versions of relevant programming libraries using existing technologies from the Score-P infrastructure. This allows for fine-grained analysis and tuning.

3.2 Application Pre-analysis

The READEX analysis strategy is based on the performance dynamics analysis capabilities of PTF, which automatically characterizes present dynamism and indicates the optimization potential. The latter gives the user an estimate of the performance and energy efficiency gains that can be realized using the READEX methodology.

In the pre-analysis step, the application is run once with a representative data set. During this run, relevant timing information is recorded. The gathered performance data reveals application regions with run-times above a certain threshold. The instrumentation of fine-grained regions with run-time below the threshold is then removed, e.g., through the definition of an instrumentation filter, to reduce the perturbation of the application. This step prepares the application for all following steps of the methodology.

In a second iteration, the application is run again with the same, or extended, data set and relevant performance and energy metrics are collected. This results in time-series of measurements representing temporal evolution of each region's computational characteristics over multiple application phases. Similarly to what is described in the use-case scenario section of Chap. 2, each region's execution, which is represented by a point in this space, corresponds to a run-time situation (RTS). A first analysis reveals whether relevant code regions exist that exhibit enough dynamism to make the following tuning steps worthwhile. The dynamism can result from differences in compute- and memory-bound operation, either between regions or between different runs of a given region. Regions found to have such dynamism are defined to be significant, and will be tuned in the following design steps. If no such dynamism is detected the tuning should be aborted due to homogeneous application behavior. Since the run-time tuning necessarily causes an overhead, a minimum dynamism threshold is defined to ensure a satisfactory overall gain.

3.3 Derivation of the Tuning Model

In order to build a tuning model that guides the adaptation of the system (both application and platform) to the dynamically changing requirements, PTF, Score-P, and the later described READEX Run-time Library are used to perform an automatic search for optimal system configurations for the significant regions identified in the previous step. To configure the platform according to each experiment, the READEX Run-time Library (RRL) is used. Details regarding the RRL are given in Sect. 3.4.

Exploration of the space of possible tuning configurations is controlled by PTF tuning plugins. Each tuning plugin is responsible to tune a specific aspect, which can include one or multiple related tuning parameters. READEX has developed and supports a number of plugins for hardware, system software, and application aspects.

The architecture of the design-time tools required to explore optimal system configurations is depicted in Fig. 6.2. To determine the optimal platform configurations, PTF runs and measures the instrumented application. Here, various system configurations are evaluated in terms of the requested objective functions for the identified RTSs. Afterwards, the results are stored in the RTS database.

Since the search space for optimal configurations is potentially large, a number of possible search strategies were identified as part of the project. This includes heuristics based parameter selection, inter-phase comparisons with an underlying

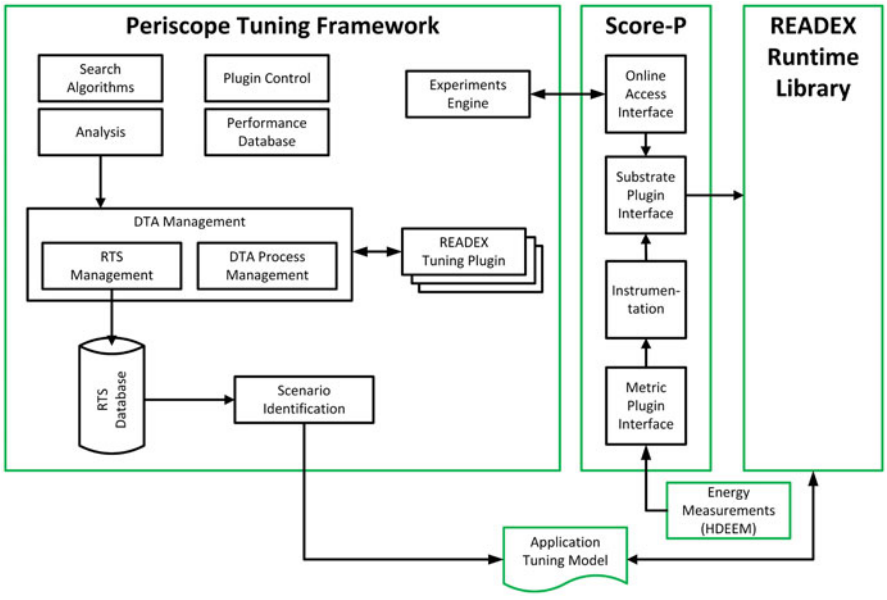


Fig. 6.2 Architecture of the design-time part of the READEX tool suite

approximation of the expected objective function, or a simple comparison with the objective function values taken during a baseline run.

After all relevant system configurations are evaluated for all RTSs, a scenario identification module groups RTSs into a limited number of scenarios, e.g., up to 20. Each scenario is represented with a common system configuration. The bound on the number of scenarios limits the frequency and associated overhead from configuration switching at run-time. As in all system scenario approaches, it is necessary to predict upcoming scenarios at run-time. This is done using the provided identifiers. An RTS is mapped to a scenario based on its signature, i.e., the current identifier values. An example of such an identifier is the call-path that has been followed to reach the current region. A configuration selector is also generated. In its simplest form, this is a function returning a set of tuning parameter values according to a one-to-one mapping between scenario and configuration. A given scenario can, for example, directly be used to specify the voltage and frequency setting to apply. The set of scenarios, the RTS signatures pointing to the scenarios, and the system configuration of each scenario, are stored in the form of a serialized text file as an application tuning model (ATM), to be loaded and applied during production runs at run-time.

3.4 Run-Time Application Tuning

Once the analysis is finished and the ATM is created, the application can be optimized in production. For this, the READEX Run-time Library (RRL) is used. It connects to the performance measurement infrastructure Score-P via a plugin interface [17], and reuses the existing code instrumentation from design-time tuning. The RRL reads the previously obtained knowledge about application dynamism, stored in the ATM, to optimize the application's energy consumption. For already seen RTSs, the optimal configuration is directly extracted from the tuning model. For unseen RTSs a calibration mechanism is used.

The architecture of the RRL is depicted in Fig. 6.3. Note that even though it is included in the figure, PTF is not used during production runs. As mentioned in Sect. 3.3, the RRL is used at design time to set the system configuration during the search PTF performs for optimal configurations. At run-time, the RRL obtains the scenarios with configurations from the ATM, which were generated by PTF at design-time.

A production run of an application starts with loading the ATM into the tuning model manager (TMM). When a new region is entered during the application execution, Score-P notifies the control center, which passes the information to the RTS Handler. The RTS Handler then checks if the region is significant and gets the best configuration for the current region from the TMM. Finally, the RTS Handler passes this configuration down to the parameter controller, which configures the different parameter control plugins (PCPs). PCPs are responsible for setting different system configurations like the CPU frequency or the number

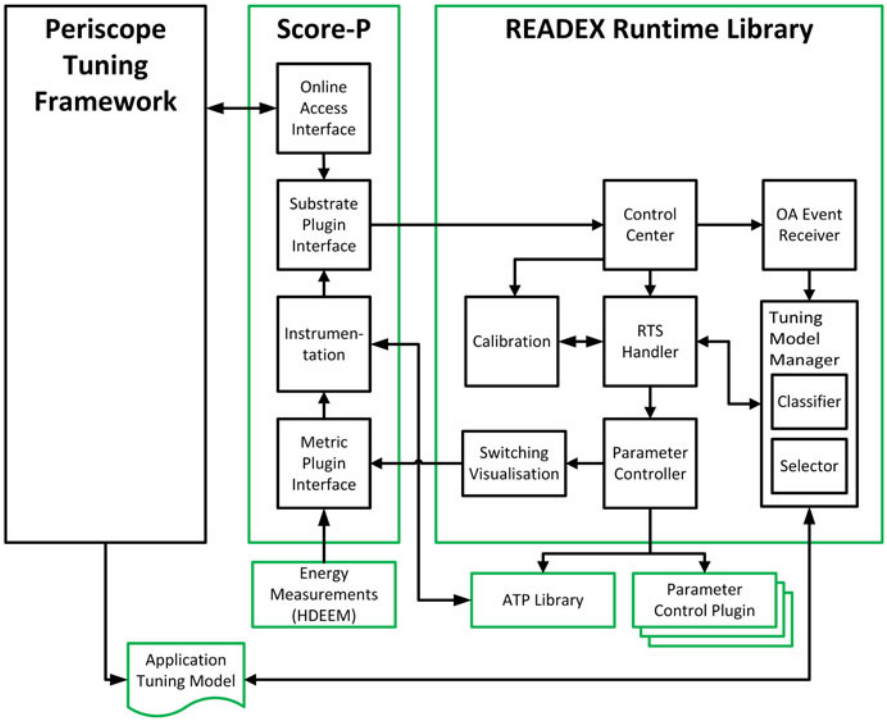


Fig. 6.3 Architecture of the READEX Run-time Library (RRL)

of OpenMP threads. They are employed both during the derivation of the tuning model at design-time and during a production run.

During the application run, a calibration mechanism can handle unseen RTSs and changes to the environment. The calibration is based on established machine learning approaches, the details of which are outside the scope of this chapter. Once the calibration detects a new optimal configuration for a certain scenario, the tuning model is updated. This leads to a constantly improving tuning model.

4 Experiments

The READEX project considers two different metrics for evaluation of the project success: the achieved improvement in energy efficiency compared to the default system configuration, measured in energy-to-solution, and the time and effort required to achieve this improvement compared to performing manual tuning.

For evaluation and validation of the project results, READEX has employed a co-design process in which the auto-tuning methodology and the tool suite have

```
int main( int argc, char ** argv ) {

    // Initialize application

    // insignificant region
    read_mesh( ... );
    // significant region, compute intensive
    assemble_Vh( ... );
    // significant region, compute intensive
    assemble_Kh( ... );
    // significant region, memory bound, MKL NUMA effects
    gmres_solve( ... );
    // significant region, I/O bound
    print_output( ... );

    // Finalize application

    return 0;
}
```

Listing 6.1 BEM4I sketch

been developed in parallel with a manual tuning effort of selected applications and computational libraries. The information exchange that resulted from this strategy has benefited both, the creation of the tools-aided methodology, and the manual tuning efforts.

The goal of this case study is to show how application dynamism can be exploited to improve energy efficiency. The evaluation was done on the *taurus* system installed at Technische Universität Dresden. The system is equipped with more than 1400 power-instrumented nodes with two 12-core Intel Xeon E5-2680v3 (Haswell-EP) processors each. The power-instrumentation allows for scalable and accurate energy measurements with a fine spatial and temporal granularity (CPU, memory, and whole node with up to 1000 Samples/s) [9]. The tests were performed on a single compute node. Hence, overall interconnect and distribution of processes in the network do not influence the results. To further validate the READEX methodology in the HPC environment the tool suite has also been tested on massively parallel applications. Scalability experiments with the ESPRESO library developed at IT4Innovations are provided in [16] but are outside the scope of this chapter.

Partial differential equations (PDEs) are often used to describe phenomena such as sheet metal forming, fluid flow, and climate modeling. One of the numerical approaches to solve PDEs is the boundary element method (BEM) as implemented, for example, in the BEM4I library [13]. In contrast to volume based methods, such as the finite element/differences/volume methods, BEM gives dense matrices whose assembly (V_h, K_h) results in compute bound code. This fact is even more pronounced when the assembly kernels are parallelized and vectorized as in the case of BEM4I [14, 21]. However, as shown in Listing 6.1, BEM4I also uses the iterative GMRES solver. This solver is based on the matrix–vector product as implemented

in the Intel Math Kernel Library (MKL), which tends to be less compute intensive and results in memory bound computation. Furthermore, printing the results for visualization leads to an I/O bound region. This dynamic behavior makes the BEM4I library suitable for testing the READEX methodology. The application was compiled with the Intel 2017 compiler with `-O3 -xcore-avx2` flags to enable the AVX2 vector instruction set available in the Haswell CPUs. The environment variable `KMP_AFFINITY` was set to `compact,granularity=core`.

For the experiment, both manual and READEX assisted tuning were performed. Listing 6.1 shows the relevant regions to be the assembly of two system matrices V_h, K_h, the iterative GMRES solver, and the I/O region, all included in a single non-iterative phase region. The HW and run-time parameters that were analyzed include the core/uncore frequency (ranging between 1.3 and 2.5 GHz/1.2 and 3.0 GHz, respectively) and the number of OpenMP threads (ranging from 4 to 24). The results that were obtained by automatic instrumentation with the READEX tool suite, i.e., application pre-analysis to detect significant regions, PTF (generation of the tuning model), and RRL (dynamic run-time switching), are compared with results obtained using the MERIC tool (tuning model and dynamic switching) developed at IT4Innovations for the purposes of manual application tuning. Note that READEX automatically identified all regions previously annotated manually for MERIC. With such an automatic instrumentation, tuning effort is decreased significantly.

Table 6.1 presents the HW and run-time configurations obtained by MERIC, namely the default configuration, the static optimum applied for the whole phase, and the optimal setting for each annotated region. It can be seen that the analysis leads to expected results with high core frequency, low uncore frequency, and a high number of threads for the compute bound assembly (V_h and K_h). For the memory bound solver (GMRES), the manual tuning results in a low core frequency, high uncore frequency, and the use of eight threads. The latter overcomes non-uniform memory access (NUMA) effects of the dual socket computational node. While static savings reach 15.7%, the dynamic switching among individual configurations increases the savings to 34.1%.

The results obtained by the READEX tools are given in Table 6.2. In comparison to the default, the READEX approach was able to achieve 34.0% energy savings.

Table 6.1 Optimal frequencies and energy saving for the BEM4I solver (MERIC)

HW conf. unit	Region –	Core freq. GHz	Uncore freq. GHz	Threads –	Energy saving %	Time saving %
Default	Phase	2.5	3.0	24	–	–
Static	Phase	2.5	2.2	16	15.7	−6.2
Dynamic	V_h	2.5	1.4	24	–	–
	K_h	2.1	1.4	24	–	–
	GMRES	1.7	2.2	8	–	–
	I/O	2.5	2.2	4	–	–
	All	–	–	–	34.1	10.9

Table 6.2 Optimal frequencies and energy saving for the BEM4I solver (PTF + RRL)

HW conf. unit	Region –	Core freq. GHz	Uncore freq. GHz	Threads –	Energy saving %	Time saving %
Default	Phase	2.5	3.0	24	–	–
Dynamic	V_h	2.5	1.4	24	–	–
	K_h	2.1	1.4	24	–	–
	GMRES	1.7	2.2	8	–	–
	I/O	2.5	3.0	8	–	–
	All	–	–	–	34.0	10.9

Note that the fully automatic READEX tool suite is able to reproduce the optimal settings from the manually tuned application.

Although this is not common for a general application, the tuning of energy also leads to the decrease in run-time in the case of BEM4I. This is caused by NUMA effects of the MKL solver—the tuned version runs on eight threads and due to the compact affinity all threads run on a single socket.

5 Conclusions

Energy efficiency and extreme parallelism are the major challenges on the road to Exascale computing. The European Union Horizon 2020 project READEX has addressed these by providing application developers with a tools-aided methodology for a combined design-time/run-time approach for dynamic adaptation to changing resource requirements. This will significantly improve energy efficiency and performance by exploiting the resources available to the application while reducing the programming effort through the automation.

In order to achieve its ambitious goals, the project has been based on two proven technologies: the system scenarios methodology as presented in this book and the static auto-tuning known from the HPC community. This chapter has presented the main concepts being used in the READEX project, as well as experimental results demonstrating the type of dynamic behavior the project will exploit.

Contemporary and future embedded systems experience continuously increasing computational capacities, e.g., through the use of heterogeneous many-core platforms. Therefore, techniques developed in the READEX project, e.g., for auto-tuning, parallel decision making, and run-time calibration, can be transferred back into this domain.

Acknowledgements The research leading to these results has received funding from the European Union's Horizon 2020 Programme under grant agreement number 671657.

References

1. S. Benkner et al., PEPPHER: efficient and productive usage of hybrid computing systems. IEEE Micro **31**(5), 28–41 (2011)
2. S. Benkner, F. Franchetti, H.M. Gerndt, J.K. Hollingsworth, Automatic application tuning for HPC architectures (Dagstuhl Seminar 13401), in *Dagstuhl Reports*, vol. 3, no. 9, pp. 214–244, 2014, http://drops.dagstuhl.de/opus/volltexte/2014/4423
3. E. César, A. Moreno, J. Sorribes, E. Luque, Modeling master/worker applications for automatic performance tuning. Parallel Comput. **32**(7), 568–589 (2006)
4. European Union FP7 project 248481, Automatic online tuning (AutoTune), http://www.autotune-project.eu/. Accessed 25 Nov 2016
5. European Union FP7 project 248647, ENabling technologies for a programmable many-CORE (ENCORE), http://cordis.europa.eu/project/rcn/94045_en.html. Accessed 26 Mar 2018
6. European Union Horizon 2020 project 671657, Run-time exploitation of application dynamism for energy-efficient exascale computing (READEX), http://www.readex.eu. Accessed 11 Feb 2019
7. I. Filippopoulos, F. Catthoor, P.G. Kjeldsberg, Exploration of energy efficient memory organisations for dynamic multimedia applications using system scenarios. Des. Autom. Embed. Syst. **17**(34), 669692 (2013)
8. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, K. De Bosschere, System scenario based design of dynamic embedded systems. ACM Trans. Des. Autom. Embed. Syst. **14**(1), article 3 (2009)
9. D. Hackenberg et al., HDEEM: high definition energy efficiency monitoring, in *Energy Efficient Supercomputing Workshop, E2SC*, New Orleans, USA, 2014
10. P.G. Kjeldsberg, A. Gocht, M. Gerndt, L. Riha, J. Schuchart, U.S. Mian, READEX: Linking two ends of the computing continuum to improve energy efficiency in dynamic applications, in *Design Automation and Test in Europe Conference & Exhibition, DATE 2017*, Lausanne, Switzerland, March 2017
11. A. Knüpfer et al., Score-p: a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir, in *Tools for High Performance Computing 2011*, ed. by H. Brunst, M. Müller, W.E. Nagel, M.M. Resch (Springer, Berlin, 2012), pp. 79–91
12. Z. Ma et al., *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogenous Platforms* (Springer, Dordrecht, 2007). ISBN 978-1-4020-6328-2
13. M. Merta, J. Zapletal, BEM4I, in *IT4Innovations National Supercomputing Center*, 2013, http://bem4i.it4i.cz/
14. M. Merta, J. Zapletal, J. Jaros, Many core acceleration of the boundary element method, in *High Performance Computing in Science and Engineering: Second International Conference, HPCSE 2015, Soláň, Czech Republic, May 25–28, 2015, Revised Selected Papers* (Springer, New York, 2016), pp. 116–125
15. R. Miceli et al., Autotune: a plugin-driven approach to the automatic tuning of parallel applications, in *Applied Parallel and Scientific Computing*. Lecture Notes in Computer Science, ed. by P. Manninen, P. Öster, vol. 7782, pp. 328–342 (Springer, Berlin, 2013)
16. L. Riha, M. Merta, R. Vavrik, T. Brzobohaty, A. Markopoulos, O. Meca, O. Vysoocky, T. Kozubek, V. Vondrak, A massively parallel and memory-efficient FEM toolbox with a hybrid total FETI solver with accelerator support. Int. J. High Perform. Comput. Appl. **33**(4), 660–677 (2019)
17. R. Schöne et al., Extending the functionality of score-p through plugins: interfaces and use cases, in *Tools for High Performance Computing 2016*, ed. by C. Niethammer et al. (Springer, Berlin, 2017), pp. 59–82

18. C. Silvano et al., The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems, in *Proceedings of the ACM International Conference on Computing Frontiers, CF '16* (ACM, New York, 2016), pp. 288–293
19. The OmpSs Programming Model, https://pm.bsc.es/ompss. Accessed 25 Nov 2016
20. A. Tiwari, C. Chen, J. Chame, M. Hall, J.K. Hollingsworth, A scalable auto-tuning framework for compiler optimization, in *IEEE International Parallel & Distributed Processing Symposium. IPDPS 2009*, pp. 1–12, 2009
21. J. Zapletal, M. Merta, L. Maly, Boundary element quadrature schemes for multi- and many-core architectures. Comput. Math. Appl. **74**(1), 157–173 (2016)

Chapter 7
System Scenario Application to Dependable System Design

Nikolaos Zompakis, Dimitrios Rodopoulos, Michail Noltsis, Francky Catthoor, and Dimitrios Soudris

1 Motivation and Related Work Comparison

Reliability is one of the most critical issues in digital design for ensuring functionality in a least-intrusive way. Operational errors can be induced by stochastic factors like random manufacturing defects, aging, and gate oxide break down effects or temporary particle interference. The unpredictable nature of such events leads to the deployment of reliability, availability, and serviceability (RAS) mechanisms that automatically correct the potential errors, ensuring the system's error-free operation. For the first time, IBM introduces the acronym RAS as a term to describe the robustness of a system in maintaining its availability and data integrity [15]. Considering that the shrinking of device dimensions has rendered the threat of failure manifestation an essential design aspect of a system, several approaches have proposed different RAS mechanisms, targeting either processor or memory failures. In this direction, the exploration of error-correction codes (ECC) [10, 11] is widely acceptable as an approach for recovering memory errors. Additionally, memory sparing and mirroring [8] techniques are also popular solutions that exploit hardware redundancy to ensure reliability. Respectively, transient errors on the processor pipeline can be easily mitigated by restarting the CPU, activating an instruction

N. Zompakis (✉) · M. Noltsis · D. Soudris
MicroLab-ECE-NTUA, Athens, Greece
e-mail: nzompakis@microlab.ntua.gr; michailnoltsis@microlab.ntua.gr; dimitriossoudris@microlab.ntua.gr

D. Rodopoulos
IMEC, Leuven, Belgium
e-mail: dimitriosrodopoulos@imec.be

F. Catthoor
IMEC and KU Leuven, Leuven, Belgium
e-mail: franckycatthoor@imec.be

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6_7

rollback technique as mentioned in [13]. Each RAS instantiation has a timing impact due to the required cycles needed to detect and recover the functionality errors which depend on the failure type as well as the RAS mechanism itself. Related work [5] has quantified this temporal overhead, introducing the term of performance vulnerability factor (PVF).

Applications' deadline restrictions demand the elimination of the PVF effects apart from correcting functional errors. The definition of a frequency operation guardband that ensures a positive slack that absorbs the timing overhead of extra cycles due to such events is a straightforward solution that significantly increases energy demands, but also prohibitive resource-consuming solution. Due to performance fluctuations, the required guardband significantly increases the energy cost. Though recent approaches [17] have improved the guardband margins, such solutions remain ineffective, especially for resource-restricted devices. Hence, flexible approaches that detect the RAS interventions and take reactions to mitigate the performance degradation should be studied. In this direction, the exploitation of a dynamic voltage and frequency scaling (DVFS) technique to ensure performance seems to be promising. Existing work [18] has developed a DVFS energy-aware reliability technique investigating static and dynamic management schemes, applying an earliest-deadline-first (EDF) scheduling. In fact, optimizing performance while applying dynamic thermal management via task mapping in many-core systems has been a thoroughly examined topic [3, 7]. In addition, studies exist focusing on both thread-mapping methodologies and DVFS algorithms that evaluate the use of the latter in real workload scenarios, targeting mostly low-power concerns [6]. Other studies [2] combine DVFS and task mapping to provide trade-offs between performance and reliability. In the domain of mixed criticality systems, task migration is a popular technique [14] to develop fault-tolerant systems that react based on the time criticality of the running task.

The majority of performance and resource-aware reliability approaches in literature can be included in one of the previous indicative solutions, including the guardband technique. The current study attempts to exploit the system scenario concept and solve performance dependability issues. In addition, we present an adaptive approach that imports innovative scheduling schemes, new to the system scenario domain. *The novelty in our implementation lies on identifying norms of RAS performance interference based on the running application situations (RTSs), enabling a partly proactive mitigation approach.*

In a more elaborate comparison, we can distinguish some common characteristics between the aforementioned approaches that are differentiated from our implementations, which provides improved design perspectives. The existing approaches (1) activate system reactions in a task-level granularity, unable to exploit energy and performance margins in the intra-task level, (2) follow a worst-case deadline (WCET) approach in respect with the scheduling decision less in the study [2] that exploits a selective task replication technique that adjusts the decision for each task considering the input/output dependencies without avoiding WCET overestimations in intra-task level, (3) utilize heuristics to dynamically manage system slack, and (4) provide mostly reactive rather than proactive capabilities, expect from study [2] that incorporates soft-error tolerance exploiting a genetic algorithm. Respectively,

the current study: (1) schedules intra-task scenario-based [4] reactions, relying on run-time system variability and thus exploiting resource trade-offs in a more efficient manner, and being more efficient and broadly applicable, (2) based on the scenario approach overcomes the constraints of a greedy WCET over-dimensioned scheduling [4], achieving better resource exploitation, (3) instead of having heavy computational heuristics to restrain system slack, a design-time analysis in RTS-level permits a more accurate and cost-efficient identification of the operation variability, and (4) incorporates proactive characteristics via the adjustment to the estimated norms of the performance fluctuations, instead of relying on heuristic algorithms (as in [2]). Additionally, we see that the former [14] is applied only on multi-core architectures, a constraint that does not exist in our proposed scheme. Lastly, compared to existing work in the system scenario scheduling [4], we add the dependability context through substantial extensions.

2 A Scenario-Based Performance Dependent Solution

Considering the existence of error-correction mechanisms that intervene to fix errors consuming extra cycles without warnings, the design of a reaction mechanism that boosts the system operation to equalize the extra delay seems a reasonable solution. The only way to speed up the operation of a digital system is to increase the operation frequency. Thus, the exploiting of several DVFS schemes is the only option. But an increase of the operation frequency can also increase the power consumption and the thermal impact on the chip downgrading the energy profile. So such actions have to be targeted and consistent for short time durations corresponding to transient RAS events. In this direction, the system scenario can organize the DVFS reactions.

The identification of several cases of RAS interfering cycles triggers corresponding scenarios that apply respecting DVFS schemes. Considering the random nature of the interfering cycles and the quantization of the scenario clustering, the existent of a PID controller with a feedback loop ensures a smooth and consistent operation. Thus, the PID controller cooperates with a scenario mechanism as follows. The incoming RTSs that are identified from a scenario detector as pieces of system execution are valued at a certain cycle budget through a scenario classification that is used as reference for the expected operation cycles. Cycle noise being superimposed on cycle budget introduces negative slack that pushes the PID controller to a new DVFS boosting. The open issue is how the PID controller can act proactively. Due to the RAS intervention randomness, the deployment of prediction mechanisms that fully forecast the interfering noise at run-time is neither realistic nor cost-effective. The key point is how to encapsulate in the reference cycle budget the gradual and partly predictable performance of the platform. The goal is to provide a low-cost run-time cycle budget mechanism that will provide opportunities for less slack without eliminating it.

Outlining, the slack reclaiming mechanism exploits a system scenario detector and the system scenarios provide the default cycle budget of the running RTSs.

System scenarios are clusters of RTSs with similar cycle budgets based on the principle of the cost similarity that is followed in any scenario application. The hierarchy and the structure of the RTSs in the system scenarios is the key for an efficient PID implementation. The effects of hardware degradation increase RAS interventions so a static definition of system scenarios and an equally static cycle noise formulation are not expected to yield dependable performance in an efficient way. A more detailed description of the scenario-based PID controller follows at the next subsection.

2.1 Scenario-Based PID Controller

A system scenario approach as refereed utilizes a PID controller with the mission to mitigate performance degradation effects by applying suitable DVFS schemes. The controller traces the time delay due to the RAS interventions, exploiting a scenario detector. The latter provides the expected execution cycles for each run-time situation (RTS) of the application, namely the cycle budgets ($\hat{R}\,[n]$). An excess of a cycle budget is identified as a RAS intervention. The application RTSs have been analyzed at design-time and have been clustered in a number of scenario cycle budgets as later explained. During the application's execution, the scenario detector identifies the current RTS, monitoring specific application RTS-related variables [4]. The PID controller recognizes the deviation from the scenario budget as a RAS intervention and triggers a respective DVFS scheme that equalizes the produced time slack. The slack for an RTS ($s\,[n]$) with a given cycle budget is the time deviation between a reference frequency (m_{ref}) operation and the current operating frequency (m_n). Figure 7.1 depicts the functionality of the proposed PID controller which is thoroughly discussed in previous work [12] and although exploits cycle budgets does not include the scenario concept. Table 7.1 explains the involved variables. The aim of the PID controller in each identified scenario is to eliminate either the negative slack (due to RAS intervention) or the positive slack (due to frequency over-boosting).

The accurate identification of the running slack is critical to achieve optimal PID responses and is directly correlated to the scenario definition. In this point, it is important to examine in more details the meaning of the RTS and the system

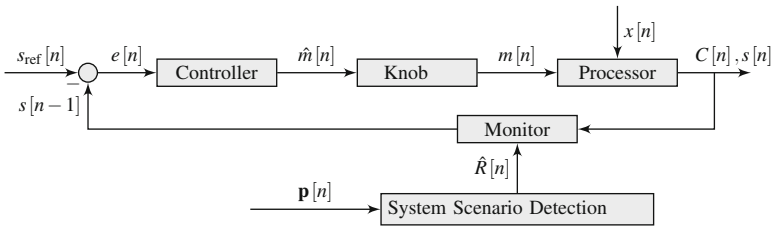


Fig. 7.1 Block diagram of the PID controller [12]

Table 7.1 Variables of the proposed controller [12]

$s_{ref}[n]$	Target slack specification (0 unless else specified)	$x[n]$	Timing noise interfering with RTS
$s[n-1]$	Slack at the end of the previous RTS	$C[n]$	Actual budget of current RTS
$e[n]$	Error signal $s_{ref}[n-1] - s[n-1]$	$s[n]$	Slack at the end of the current RTS
$\hat{m}[n]$	Proposed frequency multiplier	$\hat{R}[n]$	Default timing budget for current RTS

scenarios. An RTS is a deterministic piece of system execution with constant cost dimensions that can be used as an execution unit [4]. The wealth of RTSs in a real system is produced by any source of variation such as conditional branches in functional level or workload fluctuations. In our analysis, each RTS respects a timing deadline, as indicted by a default cycle budget and the applicable reference frequency (m_{ref}). Each system function corresponds to a sequence of RTSs. In fact, we can easily expand our definition so that a sequence rather than one RTS respects a timing deadline. However, this is outside the scope of our implementation since we focus on working on RTS-level. The run-time identification of the RTSs would provide accuracy to the estimation of the expected application's execution time in the PID controller (in Fig. 7.1). Nevertheless, the surplus of RTSs in real applications renders their exploitation as cycle budget references prohibitive. The system scenario approach [4] overcomes this issue by replacing the RTSs with representative cases called system scenarios, paying a price in accuracy. Assuming a workload of A different RTSs, each with cycle budget N_i, where $i = 0, 1, \ldots, A$, the goal of the methodology is to narrow down to B scenarios, each with cycle budget estimators $\hat{N}_j$, where $j = 0, 1, \ldots, B$ and $A >> B$. The detection of the cycle budgets can be implemented in a cost-efficient way in hardware or in software, replacing RTSs with scenarios.

The key point in the definition of the scenarios is the efficient clustering of the RTSs into groups based on their cost similarity. Considering that each RTS has a constant cycle budget, several DVFS schemes provide different cost trade-offs. Given a number of DVFS points on a processor, we can draw an energy vs delay line for each RTS. Projection of a representative evolution of RTS instances on a E-D Pareto space creates a hierarchy of RTSs based on their position in the cost space, as shown in Fig. 7.2. RTSs closer to the start of the axes have more restricted cycle budgets. The system scenarios cluster the metric space into scenario areas, hiding the details of the individual RTSs. Each RTS in a scenario is represented by the minimum RTS cycle budget (see Fig. 7.2, RTS3) to ensure the most restricted deadline in the scenario. This is important considering that scenarios are used to estimate the reference slack (s_{ref}) in the PID controller. Moreover, this approach creates an overestimated impression of negative slack for the other RTSs in the scenario (RTS1 and RTS2) that leads to over-boosting frequencies with inefficient energy results. Characterizing RTSs under the most restricted budget (MRB) of the scenario inevitably introduces a clustering overhead (light gray and brown area, see Fig. 7.2). As clustering overhead ($Cl_{RTS[i]}$) for each RTS is defined the cost

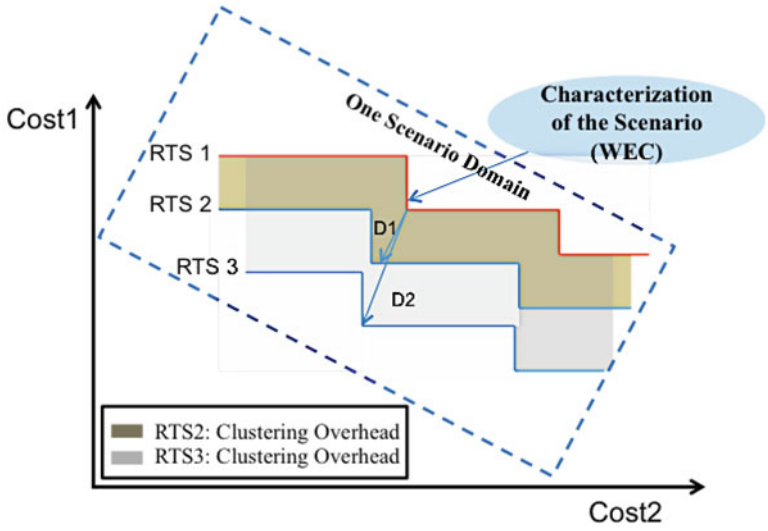


Fig. 7.2 Pareto curve hierarchy into scenarios [20]

distances (D_1 and D_2) between the original position ($OP_{RTS[i]}$) of the involved RTSs (RTS1 and RTS2) from MRB (RTS3). Equation (7.1) formulates the problem. This overhead manifests every time the RTS occurs at run-time. Thus, the total clustering overhead ($Cl\_ov_{RTS[i]}$) will be proportional to the frequency of the RTS appearance (Eq. (7.2)). $Cl\_ov_{RTS[i]}$ represents the deviation between the reference and the actual cycle budget and it is always negative since clustering tends to provide more restricted deadlines compared to the optimal case where every RTS is treated fully individually. The distance of each RTS Pareto curve expressed as clustering overhead defines the scenario accuracy and represents the first clustering criterion of RTSs into scenarios.

$$Cl_{RTS[i]} = \sum_{i=0}^{n} MRB_{RTS[i]} - OP_{RTS[i]} \tag{7.1}$$

$$Cl\_ov_{RTS[i]} = \sum_{i=0}^{n} Fr_{RTS[i]} * Cl_{RTS[i]} \tag{7.2}$$

The second important aspect to decide is the scenario switching. We prefer to keep scenarios stable as long as possible due to the transition cost between the scenarios. This cost will normally depend heavily on the initial and final state. Considering the average transition frequency ($Fr_{Sen[i \rightarrow j]}$) from scenario j to the next scenario j+1 and the respecting switching cost ($Swit_{Sen[j \rightarrow j+1]}$), the mathematical formulation of the total switching overhead ($Swit\_ov_{RTS[i]}$) is given by the following equation:

$$Swit\_ov_{RTS[i]} = \sum_{i=0}^{n} Fr_{Sen[j \rightarrow j+1]} * Swit_{Sen[j \rightarrow j+1]} \tag{7.3}$$

Optimization approaches of the aforementioned overheads tend to be competitive in respect to the targeted objectives. A decrease of the clustering overhead increases the switching overhead and vice versa. A balance between them can be achieved [20] by applying a suitable initial distribution of the RTSs into scenarios at design-time.

Therefore, the exploitation of the scenario detector allows to the PID controller to react at the presence of negative slack, enabling respective DVFS schemes. However, the time to absorb the extra cycles can create transition effects with temporary delays that can affect the stability of the application performance. The next section examines how this instability can be avoided applying revisions of the scenario sets at run-time.

2.2 Different Classes of Scenarios

Outlining the system scenario primitives, RTSs are discrete executions of TN sequences, with a default cycle budget, while the most restricted deadline into a scenario represents the fixed cycle budget for the entire RTSs. Due to variability of the RTSs into the same scenario, overestimated constraints push the PID controller to a frequency multiplier higher from what is required, thus partially mitigating noise cycles and paying a price at energy consumption. A noise shift higher than a limit creates negative slack and the PID controller is called to force to zero with a frequency increase. On condition that the RTS positions into scenarios can be updated at run-time the scenario detector can modify the RTS hierarchy by re-clustering the existing scenario distribution. This process provides a run-time adjustment of the RTS cycle budget estimations [4]. In the terminology defined in Chap. 2, this implies that we incorporate the design- and run-time calibration steps (see Fig. 2.6). This leads the PID controller to act partially proactively (see Fig. 7.3a) considering RAS events that are gradually becoming a norm on the target processor. To ensure a fully proactive operation [9] in a complete dynamic scenario approach, a predictor has to provide a run-time trend of the incoming noise. Redistributing the predicted slack across a future scheduling period in a globally optimized way as pictured in Fig. 7.3b we will achieve an even better trade-off between delay and energy. In the context of the current chapter, we focus on the scenario adaptivity through re-clustering defining the conditions of the scenario modifications at run-time. The incorporation of a predictor is not part of the current analysis and represents an individual challenging issue that needs more elaborate research that will be part of future work.

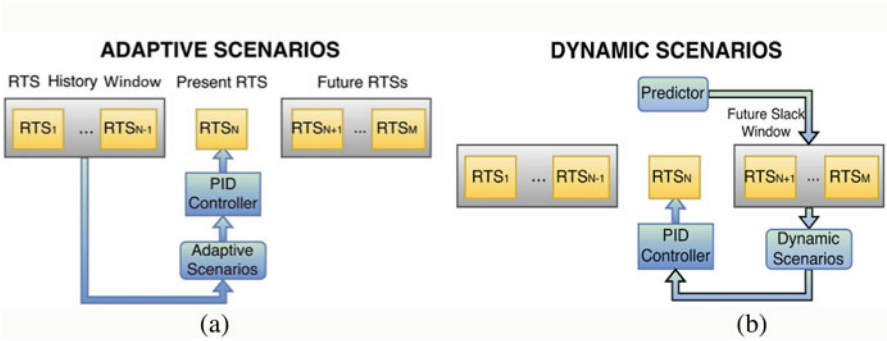
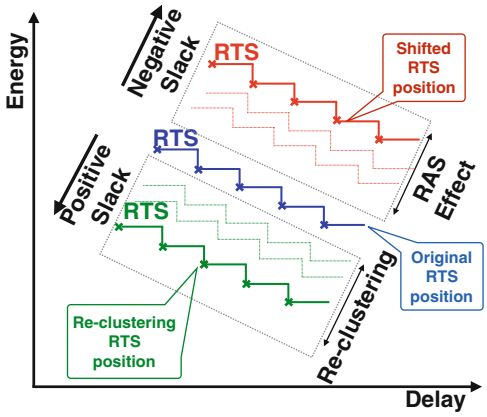


Fig. 7.3 Flow charts (**a**) adaptive vs (**b**) dynamic scenarios

Fig. 7.4 Slack impact [19]



2.3 Application of Adaptive System Scenarios

On the condition that RTS positions in the scenarios can be updated at run-time, the scenario detector can modify the RTS hierarchy by re-clustering the existing scenario distribution. This is related to the calibration step in the general system scenario approach, as outlined in Chap. 2, Sect. 5.6. This process provides a run-time adjustment of the RTS cycle budget estimations. If an RTS often appears with a specific cycle noise, this RTS fits to a new scenario by removing cycles from its budget equal to the noise. This new fit forces the PID controller to absorb the time delay before its presence. Figure 7.4 presents the two trends. The RAS interventions shift the RTS position far from the start of the axes. The re-clustering creates a positive slack (see Fig. 7.4) that works as an offset of the expected negative slack equalizing the time loss. The scope is to keep the RTS's original position stable. The RTS slack history is recorded and updated on fixed time windows, available to be exploited at run-time. The re-clustering is applied with priority in RTSs with the most intense cycle noise instances. This leads the PID controller to act partially

proactive, considering that RAS events are gradually becoming a norm on the target processor.

For each RTS re-positioning, an individual re-clustering decision aims to equalize performance degradation as described in the following sub-chapter. Considering the re-clustering decisions, the key point is to distinguish transient from (quasi-) permanent cycle noise interference events. For example, a very brief cycle noise burst that is never reoccurring should not be triggering a re-clustering process. The hardware degradation effects that occur during operation lifetime cannot be predicted in advance. Thus, an updated history trace of the slack per RTS is needed to consider the error fluctuation during execution time. The current slack ($RTS\_Slack[i]$) for a moving average window of K_w RTS executions is given by Eq. (7.4). After identifying a new trend in cycle noise manifestation, a re-clustering of the RTSs into system scenarios follows:

$$Slack_{RTS[i]} = \frac{\sum_{i=0}^{N} (RTS\_Slack[N - k_w + i])}{k_w} \tag{7.4}$$

Algorithm 1 Re-clustering algorithm

```
 1: sc_rts=detect_scenario(rts)                    ▷ scenario identification
 2: sl_rts = pid_controller(sc_rts)                ▷ slack identification
 3: if sl_rts > 0 then
 4:     if rts_list(rts) then
 5:         update_rts_slack(rts, sl_rts)          ▷ update rts slack history
 6:         if check_slack_status(rts) then
 7:             Np=new_rts_pos(rts, sl_rts)        ▷ new rts position
 8:             Sc_fit=Find_scenario(Np)           ▷ best fitting to scenario
 9:             Merge (rts, Sc_fit)                ▷ merge rts to the new scenario
10:         end if
11:     else
12:         store_rts_list(rts, sl)                ▷ new rts slack registration
13:     end if
14: end if
```

Algorithm 1 provides a brief description of the re-clustering process. For each RTS execution, its original scenario cycle budget (sc_{rts}) and cycle noise (sl_{rts}) are recorded, while the traced RTS slack is registered to a list ($rts\_list(rts)$). If the RTS already exists in the list, its record is updated ($update\_rts\_slack$). A re-clustering is triggered for a specific time window ($check\_slack\_status(rts)$) if the status of the involved RTS shows a stable behavior. Also, information about the distance from the neighboring scenarios ($Find\_scenario(Np)$) is exploited to recognize the optimal re-clustering decision, updating only the scenario data that have changed. Each re-clustering ($Merge(rts, Sc\_fit)$) is based on the new RTS position ($new\_rts\_pos(rts, sl\_rts)$) in the metric space after reviewing the noise

updated data, ensuring that the RTS deadlines in the new scenario will be respected with the minimum clustering overhead. The re-clustering is a parallel process that does not interrupt the operation of the PID controller. In any case, the re-clustering time can easily remain as fraction of the RTS operation time, due to the simplicity of the RTS-reordering algorithm (Sect. 4). At the end of each re-clustering, the PID controller triggers the suitable DVFS scheme, eliminating slack with the minimum energy cost. A trade-off between slack and energy exists that will be explored in Sect. 3 with simulation results.

2.4 Re-clustering Decisions

The re-clustering process changes the RTS hierarchy into scenarios by removing from the default RTS reference budgets the cycle noise (Eq. (7.4)), artificially creating positive slack by shifting the position of the RTSs into the scenarios (see Fig. 7.4). This creates the impression of an extra delay to the PID controller proportional to the removed cycles, triggering more aggressive DVFS schemes that absolve the negative slack. Examining the re-clustering decisions that have to be considered at run-time, we outline the following cases:

- In the first case, the scenario budget is restricted enough and no slack exists; hence, there is no need for scenario change (see Fig. 7.5a). The added noise (red shifted Pareto curve) activates a revision of the RTS budget (green Pareto curve) that does not exceed the distance between the RTS and the scenario (Sc_2), exploiting the clustering overhead as an offset. Thus, clustering overhead can be utilized as a proxy of cycle noise tolerance. A high clustering overhead permits wide RTS shifts, absorbing slack and paying the cost of overestimated DVFS schemes. The next four cases cover situations where the slack cannot be avoided.
- In the second case (see Fig. 7.5b), in order to equalize the noise delay (red Pareto curve), the revised RTS (green Pareto curve) is located just under the scenario budget SC_1. Thus, Scenario #1 represents a new more restricted case due to the RTS shifting that modifies the existing scenario budget.
- In the third case (see Fig. 7.5c), a new distinct scenario budget is created. Two reasons lead to such a decision: (1) none of the existing scenario cycle budgets cover the new RTS shift or (2) the creation of an individual scenario is more cost-effective than merging with an existing scenario.
- In the fourth case (see Fig. 7.5d), the shifted RTS (green Pareto curve) is closer to a neighboring scenario (SC_2) than to the original (SC_1). Thus, the RTS is merged into the new scenario (SC_2) achieving a more optimal fit, considering the distance.
- The fifth case (see Fig. 7.5e) is a combination of the second and the third aforementioned cases. A neighboring scenario merges the revised RTS. However, the position of the RTS in the new scenario is located just under the scenario budget, changing the new scenario cycle budget as the most restricted case.

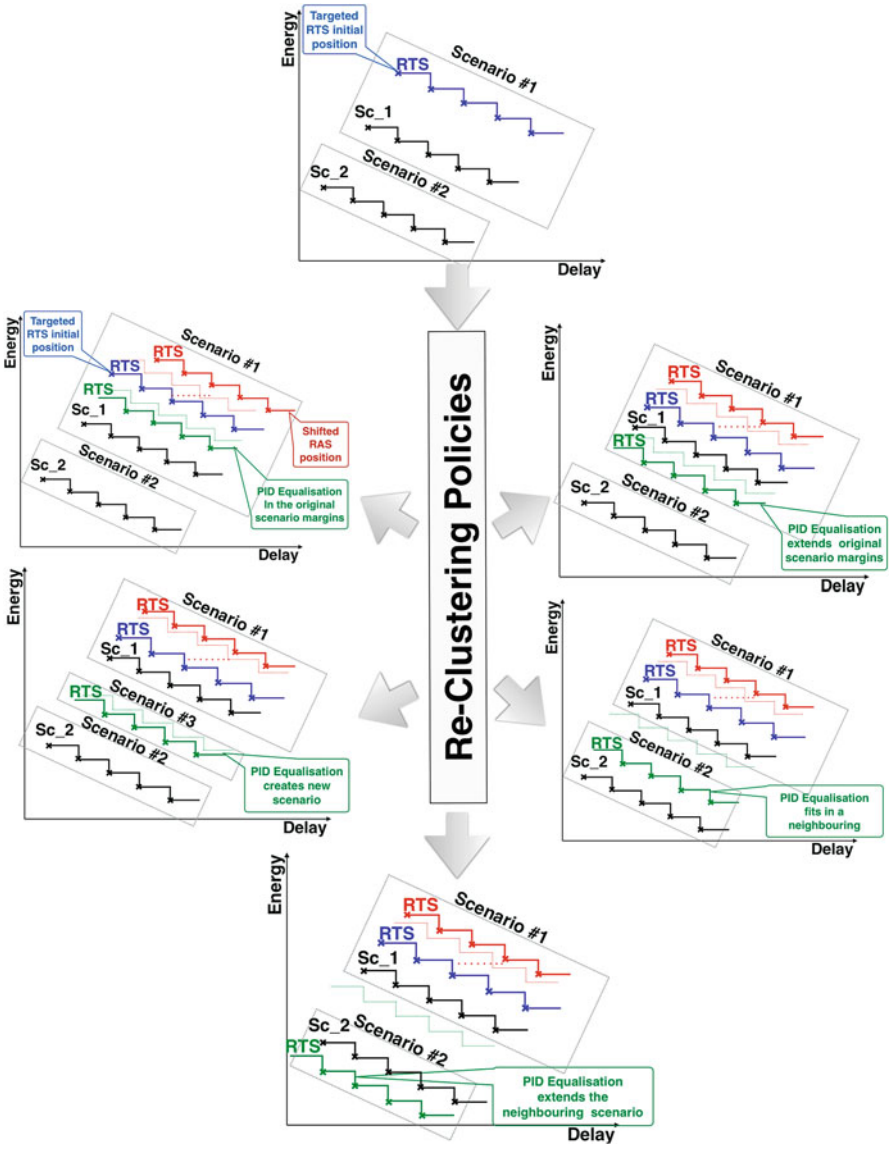


Fig. 7.5 Re-clustering policies [19]

3 Experimental Results Based on Platform Simulations

The application under study is commercially used for spectrum sensing and band allocation purposes [16], and constitutes an exploration space of about 800 RTSs, clustered in 30 scenarios that achieve an efficient initial balance between scenario

overheads [20]. We submit this workload to a simulator that has been presented in the previous work [12], to study our proposed mitigation scheme. We will focus on an implementation based on the system scenario approach with no re-clustering mechanisms and also on an implementation that incorporates the scenario adjustment concept and examine the efficiency of both configurations. For the sake of completeness, we repeat the basic functionality of the simulator here. The PID controller operates at RTS-granularity; thus, the scenario unit instantiates also the cycle budget estimator $\hat{N}_n$ in the same basis.

3.1 Platform and System Setup

Additionally, cycle noise is injected based on the values of Λ and M that represent, respectively, the rate and amplitude of RAS events in terms of clock cycles. We assume that cycle noise follows a bivariate Gaussian distribution with mean Λ and M values (λ and μ) and σ ($\sigma\{\Lambda\}$, $\sigma\{M\}$). The timing and power models of the simulator create traces from which DVF (deadline vulnerability factor) and total energy can be deduced. Given the definition of cycle budget estimators in Table 7.1 and their derivation, DVF expresses the per unit increase in execution time required for an instruction stream, following the temporal overhead of invoked RAS mechanisms. Respectively, PVF (performance vulnerability factor) expresses the per unit increase in number of clock cycles required for a specific workload, as a result of RAS technique invocation. It is noteworthy that all overheads of DVFS actuations are included in both the timing and power models. Finally, at the end of each RTS, the PID controller selects the frequency multiplier of the next iteration, as detailed in Fig. 7.1.

The simulation results presented herein correspond to a set of random streams of the target application RTS. The simulated workload assumes equally probable RTSs. Cycle noise is injected by sweeping μ and λ values, while $\sigma\{M\}$,$\sigma\{\Lambda\}$ are kept constant. The noise histogram for one stream of RTSs is shown in Fig. 7.6. In fact, each measurement corresponds to a stream of RTSs and PID controller gains are set following the tuning Ziegler Nichols method [1]. For each combination of μ and λ, we present the average over 100 random measurements to provide insight into the statistical variability of our simulation results. To highlight the benefits of the scenario concept we will examine three test-cases: (1) a 30% frequency guardband utilized to alleviate cycle noise, (2) a system scenario scheduling without re-clustering capabilities, and (3) an adaptive scenario implementation that realizes the aforementioned re-clustering schemes. A first conclusion of our experiments is that apart from extreme noise cases (high μ and low λ values), the PID controller manages to enable dependable performance, especially considering the increment in PVF (see Fig. 7.7) which is the result of cycle noise injection.

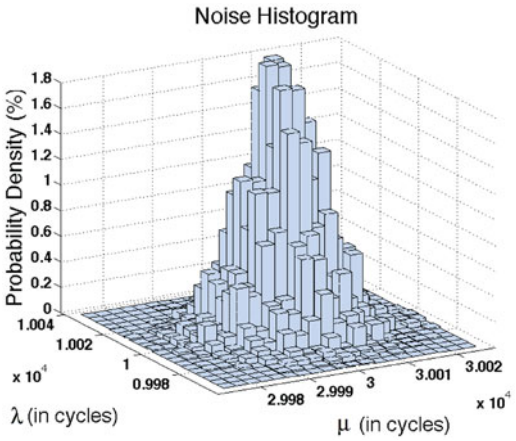


Fig. 7.6 Cycle noise for one RTS stream

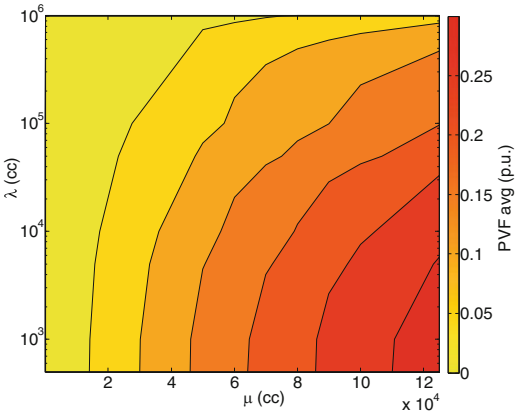


Fig. 7.7 Average PVF values for 100 iterations [19]

3.2 Dependability Results

From Fig. 7.8, we clearly distinguish the effectiveness of the adaptive scenario configuration over the system scenario one. While in the system scenario implementation, DVF seems to balance to a fixed value close to zero, in the adaptive scenarios implementation, DVF adjusts more accurately. The reader should keep in mind that since DVF describes the increase in execution time due to cycle noise, a DVF equal to zero implies that our configuration has successfully absorbed all temporal overhead with no time penalty. In fact, the timing performance gain of the adaptive scenario approach is 5% higher than that achieved by the system scenario, while it is less effective than the guardband approach only for limited and extreme noise cases.

For a transition period (0–2000 ms), the number of the adaptive scenarios is fluctuating to reach to a stable state, as shown in Fig. 7.9. During this time, the re-clustering process generates new scenarios, increasing gradually their number up

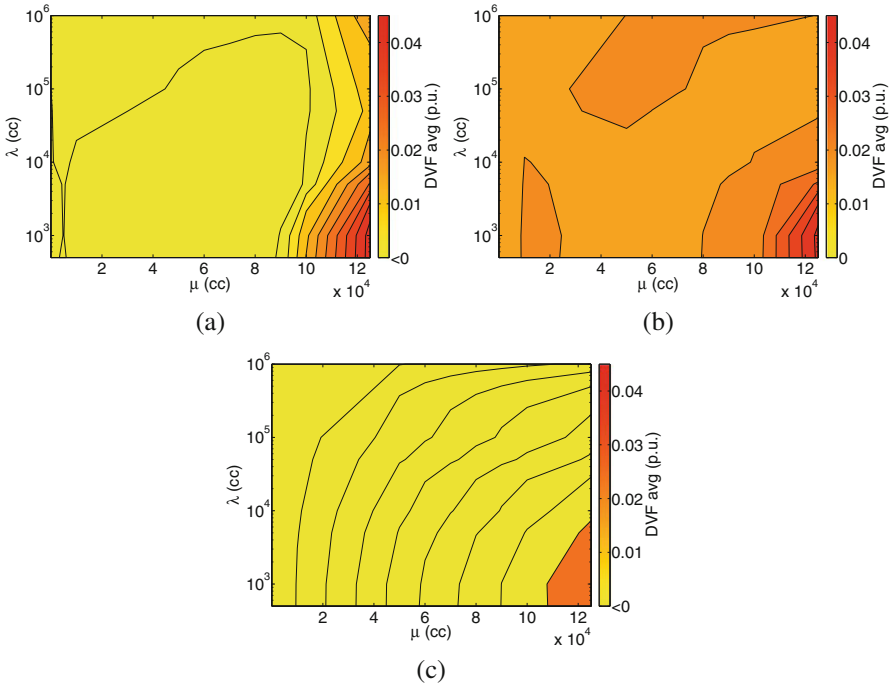


Fig. 7.8 Average DVF values for 100 iterations [19]. (**a**) DVF adaptive scenarios. (**b**) DVF system scenarios. (**c**) DVF guardband

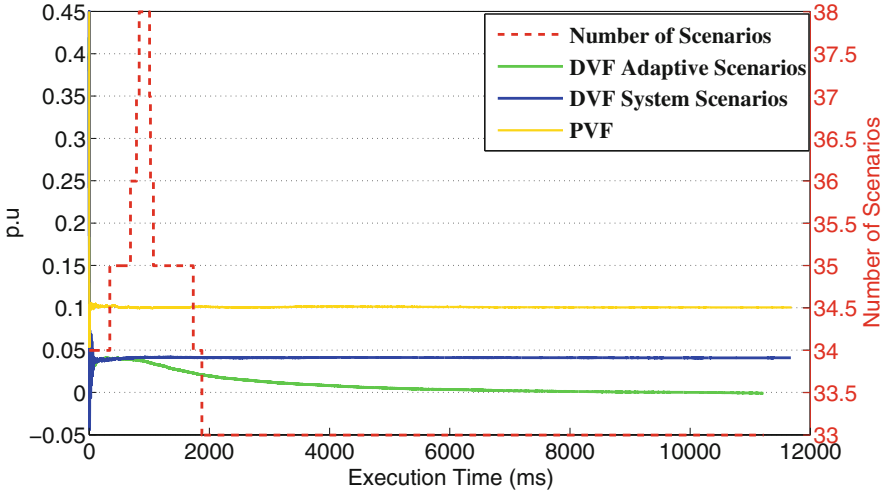


Fig. 7.9 Performance adjustability instance [19]

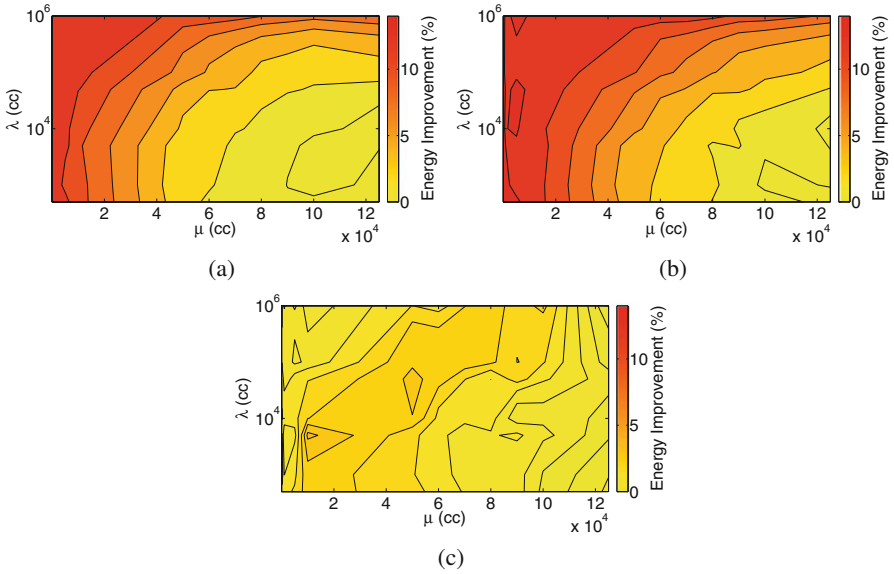


Fig. 7.10 Percentage energy deviations for 100 iterations. (**a**) Energy gain adaptive. (**b**) Energy gain system. (**c**) Energy adaptive vs system scenarios

to 38 (1000 ms) and after this point some of the existing scenarios are merged to finally reach the number of 33 (2000 ms). Thus, the adaptation process creates and merges scenarios by applying the aforementioned re-clustering decisions (Sect. 4) to find the optimum scenario hierarchy. When this is achieved (after 2000 ms), the scenarios remain stable and the DVF comes close to zero.

3.3 Energy Results

The energy gain comparing (see Fig. 7.10a,b) the two scenario approaches with the frequency guardband test-case is clearly beneficial and reaches up to 15%. The gain is decreasing as the injected cycle noise profile becomes more aggressive (high μ and low λ). This result is not unexpected, due to fact that the scenarios are stabilized to a state similar to the guardbands for extreme noise cases. Hence, without applying an over-boosting frequency policy with unnecessary energy consumption results, both scenario approaches seem to ensure performance dependability. In fact, considering the notional energy gain margins as highlighted by the energy deviation between the guardband frequency as worst-case and the reference frequency as basis, adaptive scenarios achieve an average exploitation of 83.9%, ensuring performance dependability.

The energy consumption deviation between the system and adaptive scenario is minor and does not exceed the 3% (see Fig. 7.10). This occurs due to the fact that

the extra energy consumption from frequency boosting on the adaptive scenarios is partially equalized by the less number of DVFS switches due to the adjustment. Nevertheless, a fundamental trade-off exists between the achieved performance as expressed by the DVF and the energy impact that will be examined in the next subsection.

3.4 Design Trade-Offs

The key issue in the achievement of several trade-offs is the increase of the reference cycle budgets both in system and adaptive scenarios. This is implemented by applying homogeneously a percentage relaxation factor F_{rel} to the MRB of each scenario (Fig. 7.2), having an equal clustering overhead increase in each scenario. Equation (7.5) formulates the new clustering overhead. This approach provides better energy results due to less accurate slack estimations in the PID controller and worst performance due to the increase of the clustering overhead. Figure 7.11 presents the trade-offs between system and adaptive scenarios for several relaxation factors (0–10%) and two noise instantiations. The Pareto curves of the two approaches (adaptive vs system scenarios) seem to be very close, presenting similar cost trade-offs (delay-energy) for a specific constraint area ($0.04 < DVF < 0.08$). The adaptive scenario appears just a little worse trend (longer distance from the start of the metric axes) due to the extra implementation cost that they require. We can observe that the adaptive scenario approach is of high value when strict dependability constraints are demanded. In fact, it is clear that the system scenario approach is unable to meet such demands. However, when timing constraints relax ($DVF > 0.04$), the system scenario approach appears to be Pareto-optimal.

$$Cl'_{RTS[i]} = \sum_{i=0}^{n} (1 + F_{rel}) \times MRB_{RTS[i]} - OP_{RTS[i]} \tag{7.5}$$

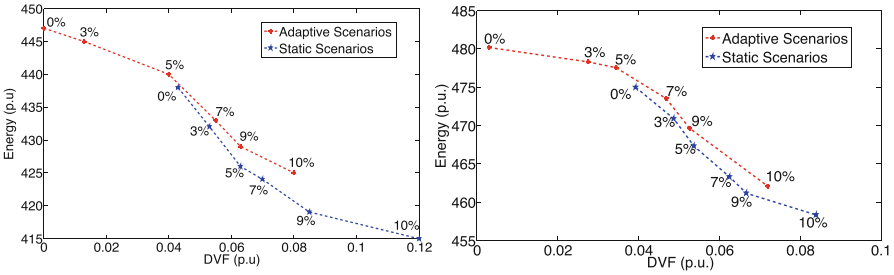


Fig. 7.11 Performance (DVF) vs energy trade-offs. (**a**) Noise #1 ($\mu = 1 \times 10^4$, $\lambda = 5 \times 10^3$). (**b**) Noise #2 ($\mu = 5 \times 10^4$, $\lambda = 5 \times 10^3$)

4 Conclusion

In the context of the current chapter, we deployed an adaptive mechanism that adjusts the system reactions in the presence of RAS interventions ensuring performance dependability. The key point is the re-clustering of the existing system scenarios with a flexible way absorbing the cycle noise that is gradually becoming a norm during the observed application operation. The experimental results prove the efficiency of the adaptive scenarios paying a low price in energy that does not exceed the 3% compared to the static system scenarios. The simulation results prove the efficiency of this approach, ensuring the system dependability constraints while achieving an energy gain up to 15% compared to a guardbanding implementation and exploiting in average 83.9% of the notional energy gain margins. Future work consists of implementing a fully proactive mechanism that will further increase performance dependability that will integrate the use of a predictor to prognosticate future slack windows and further increase performance dependability.

Acknowledgements This research is supported in part by the EU FP7 HARPA project, grant agreement 612069.

References

1. K. Astrom, T. Hagglund, *PID Controllers: Theory, Design and Tuning*, 2nd edn. (International Society of Automation, Pittsburgh, 1995)
2. A. Das et al., Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs, in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE, Piscataway, 2014), pp. 1–6
3. T. Ebi et al., Tape: thermal-aware agent-based power economy for multi/many-core architectures, in *IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009* (IEEE, Piscataway, 2009), pp. 38–43
4. S.V. Gheorghita et al., System-scenario-based design of dynamic embedded systems. ACM Trans. Des. Autom. Electron. Syst. **14**(1), 3 (2009)
5. D. Hardy et al., The performance vulnerability of architectural and non-architectural arrays to permanent faults, in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (IEEE, Piscataway, 2012), pp. 48–59
6. S. Herbert et al., Analysis of dynamic voltage/frequency scaling in chip-multiprocessors, in *International Symposium on Low Power Electronics and Design (ISLPED) ACM/IEEE*, pp. 38–43, 2007
7. A. Kumar Singh et al., Mapping on multi/many-core systems: survey of current and emerging trends, in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE* (IEEE, Piscataway, 2013), pp. 38–43
8. C.S. Mugunda et al., Improving fault tolerance. Dell Power Solutions (Aug 2005)
9. S. Munaga, F. Catthoor, Systematic design principles for cost-effective hard constraint management in dynamic nonlinear systems, in *Innovations and Approaches for Resilient and Adaptive Systems* (IGI Global, Hershey, 2013), pp. 1–28
10. G. Neuberger et al., A multiple bit upset tolerant SRAM memory. ACM Trans. Des. Autom. Electron. Syst. **8**(4), 577–590 (2003)

11. P. Reviriego, J.A. Maestro, Study of the effects of multibit error correction codes on the reliability of memories in the presence of MBUs. IEEE Trans. Dev. Mat. Rel. **9**(1), 31–39 (2009)
12. D. Rodopoulos et al., Tackling performance variability due to RAS mechanisms with pid-controlled DVFS. IEEE Comput. Archit. Lett. **14**(2), 156–159 (2015)
13. T. Sakata et al., A cost-effective dependable microcontroller architecture with instruction-level rollback for soft error recovery, in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)* (IEEE, Piscataway, 2007), pp. 256–265
14. P.K. Saraswat et al., Task migration for fault-tolerance in mixed-criticality embedded systems. ACM SIGBED Rev. **6**(3), 6 (2009)
15. D. Siewiorek, R. Swarz, *Reliable Computer Systems, 3rd edn.: Design and Evaluation* (A. K. Peters, Ltd., Natick, 1998)
16. Thales Communication and Security. Spectrum monitoring and homeland security (Aug 2015), https://www.thalesgroup.com/en/worldwide/security/spectrum-monitoring-and-homeland-security
17. V.M. van Santen et al., Designing guardbands for instantaneous aging effects, in *Proceedings of the 53rd Annual Design Automation Conference* (ACM, New York, 2016), pp. 69–75
18. D. Zhu, H. Aydin, Reliability-aware energy management for periodic real-time tasks. IEEE Trans. Comput. **58**(10), 1382–1397 (2009)
19. N. Zompakis, M. Noltsis, D. Rodopoulos, F. Catthoor, D. Soudris, Energy efficient adaptive approach for dependable performance in the presence of timing interference, in *Proceedings of the on Great Lakes Symposium on VLSI 2017* (ACM, New York, 2017), pp. 209–214
20. N. Zompakis, A. Papanikolaou, P. Raghavan, D. Soudris, F. Catthoor, Enabling efficient system configurations for dynamic wireless applications using system scenarios. Int. J. Wireless Inf. Networks **20**(2), 140–156 (2013)

Chapter 8
Scenarios in Dataflow Modeling and Analysis

Marc C. W. Geilen, Mladen Skelin, J. Reinier van Kampenhout, Hadi Alizadeh Ara, Twan Basten, Sander Stuijk, and Kees G. W. Goossens

1 Introduction

In this chapter we illustrate the application of the scenario methodology to the timed dataflow model of computation. Dataflow is an abstract mathematical model that can be used to model streaming applications and the resources on which they are realized. They are used, for instance, in the form of CSDF graphs in Chap. 4 to model applications on DVFS enabled processor platforms. Moreover, they can be effectively used to model flexible manufacturing systems, as discussed in detail in Chap. 9.

Figure 8.1 shows a dataflow model of an H.263 decoder. The circles are *actors*. They represent activities in the decoder. For example, the actor labeled vld represents the activity of the variable-length decoder. Actors execute their activities repeatedly. Actors have *dependencies* that determine their earliest activation times. An activation of an actor is called a *firing*. The dependencies are shown as incoming arrows, called *edges*, with *tokens* on them. There can be dependencies on firings of other actors, but also on earlier firings of the same actor or on external events. When an actor fires, it takes (consumes) tokens from its incoming dependencies. The firing takes a certain amount of time, after which the firing completes and produces

M. C. W. Geilen (✉) · M. Skelin · J. R. van Kampenhout · H. Alizadeh Ara · S. Stuijk · K. G. W. Goossens
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: m.c.w.geilen@tue.nl; m.skelin@tue.nl; j.r.v.kampenhout@tue.nl; s.h.seyyed.alizadeh@tue.nl; s.stuijk@tue.nl; k.g.w.goossens@tue.nl

T. Basten
Eindhoven University of Technology and ESI, TNO, Eindhoven, The Netherlands
e-mail: a.a.basten@tue.nl

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6_8

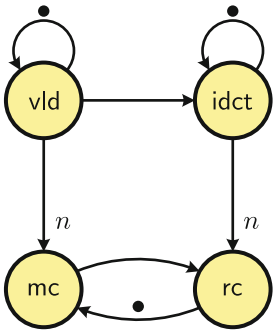


Fig. 8.1 A dataflow model of an H.263 decoder

tokens on the outgoing dependencies, thus satisfying the firing dependencies of other actors. It can also output tokens to the environment. Tokens in the model represent dependencies between firings. In the concrete systems such dependencies could be due to data dependencies between computations, but also due to resource dependencies, for example, the order of scheduling on a processor.

Different complementary views exist on dataflow models. Sometimes the functionality of actors is emphasized, where the model describes how actors compute output tokens with *values* from its input tokens with values, for example, in Kahn process networks [24]. Usually, actors represent functional, deterministic, and stateless behavior in terms of their computations. Another view on dataflow models emphasizes their timing and performance. This is the view we focus on in this chapter. In this view tokens do not carry values, but are pure dependencies. Actors have execution times that make that tokens are produced, or actors start, at certain moments in time. The timing has implications for performance properties such as throughput and latency. The models and their semantics are discussed in more detail in Sect. 3.

Dataflow models have a number of important strengths. They make *concurrency* very explicit. Actor firings that have no dependencies on one another are concurrent. The explicit concurrency can be exploited to optimize performance and for scheduling purposes. Many dataflow models (though not all) are *determinate*. In a determinate model, even though certain computations can be performed in arbitrary orders because of concurrency, the final result is independent of that order, which separates the concerns of correct functionality and scheduling. An important strength for the timed view on dataflow models is that they are *monotone*: if some input dependency is delayed (pushed into the future), then any event in the model (actor firing, token production) cannot happen earlier than its original time. Consequently, also, when an actor firing duration is shortened, no event in the model can occur later than in the original model. This property allows one to make simple, deterministic worst-case abstractions of systems, which can be analyzed very efficiently. This is discussed in more detail in Sect. 2.

Dataflow models also have some weaknesses that need consideration. The more expressive dataflow models are generally also more difficult to analyze.

Kahn process networks (KPNs) [24], for example, allow general determinate data-dependent behavior to be expressed, such as an actor that produces a different number of output tokens depending on the value of an input token. Analysis of KPN models, however, is very hard. The problem of finding minimal buffer sizes that allow deadlock-free execution, for example, is undecidable [14, 32]. At the other end of the spectrum we find dataflow models with limited expressiveness, such as synchronous datalow (SDF) [25] in which the actors are deterministic and produce and consume fixed numbers of tokens. They are much more amenable to analysis and synthesis techniques, such as optimal scheduling [38] or buffer sizing [30, 40], although these problems are still in higher complexity classes [30]. These more static dataflow models do not suffice to model modern dynamic applications without being overly pessimistic about their performance and resource usage. Because of this trade-off between expressiveness and analyzability, many slightly different dataflow models have appeared in literature [42]. In this chapter we show how the scenario methodology can be applied to dataflow models to arrive at a model that strikes a particularly useful balance between the expressiveness needed to address the dynamic variation in modern applications and architectures, to limit the overestimation of resources and to preserve significant analysis and synthesis possibilities.

Dataflow models are suitable for applications that consist of fragments of sequential behavior that are internally deterministic and static and that are composed into parallel applications that may exhibit non-deterministic variation in execution of those behaviors. The dataflow models are also mostly suitable for applications that repetitively perform the same, or similar, behavior, possibly operating on or transforming streams of data.

The H.263 application is an example of an application that exhibits significant amounts of variation. In particular, the amount of data spent to encode the difference between successive video frames strongly depends on the degree of difference between the frames. This is expressed by the number of macro-blocks used to encode the difference. In the model, this is the number of tokens exchanged between the vld and the idct actors, indicated in Fig. 8.1 with the consumption rate of n, tokens per firing, which may take different values for different frames. One can prove that a static model with a fixed number of blocks (the maximum value that n may take) is a conservative abstraction, but it is in many circumstances too pessimistic. Alternatively, a model can be made that exactly defines the number of macro-blocks and how it depends on the data input, but such a model would be too complex for efficient analysis. The scenario methodology proposes to group the run-time situations that occur in a limited number of scenarios in which the worst-case resource usage overestimates the actual usage only by a moderate amount. We do this by grouping the number n of macro-blocks into ranges within which we use the maximum as a worst-case representative. For example, the run-time situations in which there are 30–40 macro-blocks may be represented by a single scenario in which the work load is assumed to correspond to 40 macro-blocks, the worst-case of all run-situations covered by the scenario. Similarly, the actor execution times in the model are selected to correspond to worst-case execution times of the software.

Besides the scenarios due to varying number of macro-blocks, in H.263 variation occurs due to the fact that some frames are encoded independently from previous frames. We capture the behavior of that run-time behavior by a separate scenario as well.

There are many examples where the same approach applies, for example, in the channel equalizer model in [27]. In this system periodically one of every eight symbols triggers a channel estimation computation with extra computational requirements. In this case the dynamic variations exhibit deterministic periodic patterns. Cyclo-static dataflow models can be used to represent this behavior. The scenario-aware dataflow (SADF) model, discussed in this chapter, generalizes cyclo-static dataflow (CSDF).

A last example to mention is a WLAN receiver [28]. Figure 8.2 shows the finite state automaton that specifies the possible sequences of scenarios of this model. The reception of a frame consists of a sequence of different activities: synchronization, header processing, payload processing, and positive or negative acknowledgement after error-detection. We use scenarios to capture the variations in the run-time situations associated with these activities. Switches between activities (scenarios) are non-deterministic. Synchronization with the sender may be lost at any time and a frame may contain a different number of payload symbols. They can therefore not be expressed by CSDF. However, their occurrences are constrained to particular patterns. We want to be able to exploit this knowledge. The finite state automaton of Fig. 8.2 encodes the possible sequences of scenarios. Loss of synchronization causes a transition back to the synchronization state to be taken. The self-transition on the payload state is used when another payload symbol follows in the frame. When the frame ends, the scenario changes to the acknowledgement scenario. Note that it is conservative in the sense that it still allows some sequences of scenarios that cannot occur in reality. For instance, it allows for arbitrarily long sequences of payload symbols, even though under the real protocol constraints there can be no more than 256. This could be encoded in a finite state automaton, but it would have a large number of states. Possibly more compact representations could be applied such as the regular expressions used by Ara et al. [1] to obtain a compact and exact representation.

The behavior in the individual scenarios is defined by dataflow graphs. Figure 8.3 shows the behavior of the sync scenario and Fig. 8.4 shows the behavior of the

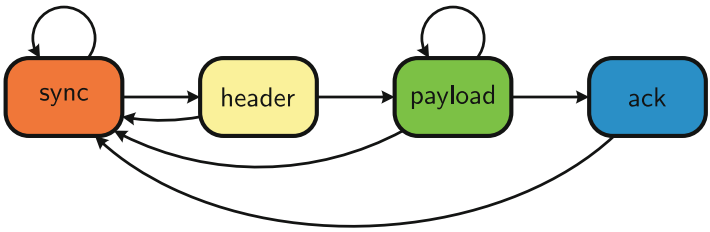


Fig. 8.2 Specification of the possible scenario sequences

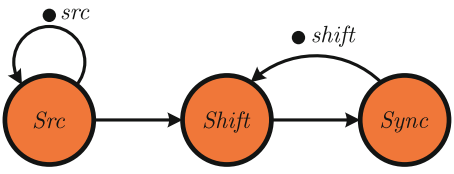


Fig. 8.3 Dataflow graph of the sync scenario

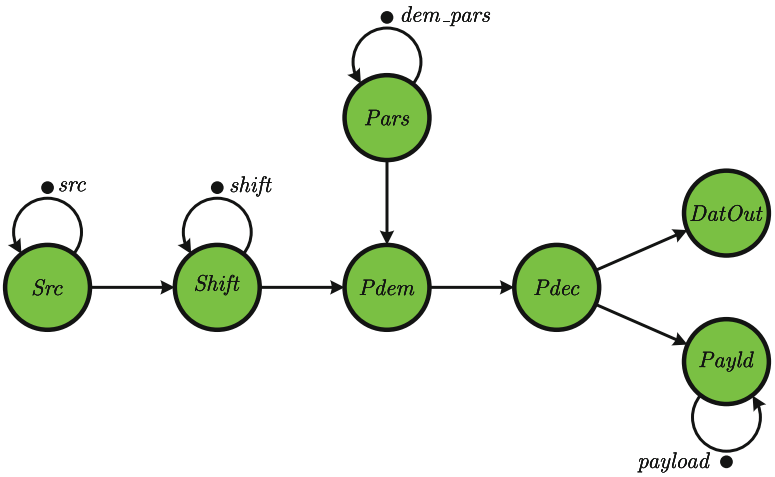


Fig. 8.4 Dataflow graph of the payload scenario

payload scenario. Other examples of applications modeled with SADF can be found in literature [1, 27, 29, 43].

2 Scenarios in Dataflow Modeling

2.1 Relation to the Scenario Methodology

The essential property of the SADF model is that it maintains as much as possible of the determinacy of dataflow behavior while introducing the possibility for non-deterministic variation in the form of scenarios as elaborated on the H.263 decoder of Fig. 8.1. Every scenario is represented by an SDF graph that represents the worst-case from a multi-dimensional cost perspective within the cluster of run-time situations it represents.

The concept of SADF lies perfectly within the general scenario methodology as discussed in Chap. 2. In particular, the system scenario methodology is a five step approach in which each step has a design-time and a run-time phase. The first step of the methodology is *identification*, performed at design-time, in which identified run-time situations are clustered into scenarios based on a particular multi-dimensional

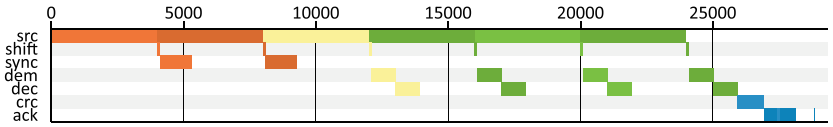


Fig. 8.5 Gantt chart of an execution of the WLAN model

cost perspective. In terms of SADF, the identification step entails the identification of SDF-like regions in dynamic systems under consideration. The so-identified static regions are called scenarios. An aspect that is particular to the dataflow approach is that such regions can span activities in both *time* and *space*. For example, in the Gantt chart of the WLAN example in Fig. 8.5 we observe that the scenarios (differently colored parts) follow a pipelined structure and overlap in time. More about methodologies that enable the identifications of these regions can be found in [20, 33].

The second step of the system scenario methodology is known as *prediction*, where a scenario has to be selected from a scenario set based on the parameters of the run-time situation. In SADF the selection options can be constrained by a model that defines the possible scenario occurrence patterns [42, 43]. In case a finite state automaton is used, the automaton is generally non-deterministic where the non-determinism may have one of the two following roles: a *descriptive* and a *constraining* role [23]. Within the descriptive role, non-determinism captures an aspect of the world that is not completely known and that behaves in an unpredictable manner. Within the constraining purpose, the non-determinism designates different possibilities for implementation. Within the system scenario methodology the descriptive non-determinism is more pronounced in the sense that it arises from the deliberated decision to ignore the facts which influence the selection [22]. In software, these facts are often values of certain control variables. In a cyber-physical system, they could be uncontrollable events. Section 4 discusses how the, in the former situation, run-time prediction activities can be made explicit if SADF is used as a programming model. In addition to non-determinism, SADF supports the specification of probability mechanisms that govern the choice between scenarios [43]. Such quantitative information about the likelihood of scenario transitions can be used for analysis purposes or to decide on scenario switching activities when cost is involved with the switching itself.

The third step is called *exploitation*. At design-time, it involves optimization that is applied based on the knowledge of the scenario structure and patterns and at run-time it is the execution of the scenario. In the SADF context, the design-time phase involves SDF-related transformations like multi-rate expansion, retiming, pipelining, and unfolding with the purpose of optimizing graph throughput, minimizing code size, mapping to processors and memory, etc. [3, 38]. The run-time phase entails scheduling and execution of the scenario SDF graph.

The fourth step of interest is called *switching*, which is the act of run-time reconfiguration to enable the execution of another scenario. In the context of SADF and software-intensive systems, each scenario in the SADF graph could,

for example, use a different mapping to processors or memory. To implement this, a run-time reconfiguration mechanism is employed that can transfer data items (tokens) and code (actors) between different memories whenever a scenario switch occurs [42]. Stuijk et al. [41] present a design flow that maps a throughput-constrained application, modeled with an SADF to an MPSoC.

The final, fifth step is called *calibration* that collects information about values of run-time parameters and further adjusts the system to optimize against a certain cost function. Calibration is only meaningful when performance constraints are soft constraints or to optimize average-case behavior in the absence of constraints. In the context of SADF, this step may have counterparts, such as calibration of worst-case executions times, but such approaches have not been worked out as yet.

2.2 Abstraction and Refinement in Timed Dataflow

We represent run-time situations in a scenario by worst-case behavior of the scenario, or a tight upper bound on its worst-case behavior. In general systems it may be difficult to identify the worst-case behavior due to complex interactions between components or the environment, or due to resource arbitration. The longest execution time of a single task may, for instance, not always lead to the longest execution time of the overall application. Such an effect is often called a *timing anomaly* [26].

Dataflow models are monotone (more generally, max-plus linear, see Sect. 3.2) and do not exhibit such timing anomalies. This has the advantage that the worst-case situation can be easily identified and corresponds to the actors taking their largest execution times. This holds within a single dataflow graph, but also compositionally, i.e., when they are placed in a context, for example, when dataflow graphs are built hierarchically in a modular fashion. If a dataflow graph consumes inputs from its environment and produces outputs to the environments, then it can be formally shown that any run-time behavior represented by the scenario performs at least as good as the scenario representative in the following sense. Each of the outputs are produced at times, no later than they are produced in the representative behavior if the inputs are provided no later than in the representative behavior. This establishes a very precisely defined, formal abstraction–refinement relation between the run-time situations in the scenario and the representative behavior [16], in which the representative behavior is the *abstraction* and the concrete run-time situations are *refinements* of that behavior.

An important property of the abstraction–refinement relation is that non-deterministic systems (executions may vary non-deterministically under data or resource dependencies) can have deterministic abstractions (a dataflow model with fixed, deterministic, worst-case execution times). This has significant advantages for performance analysis. Only a single, deterministic behavior (the abstraction) needs to be verified and if it satisfies its performance requirements, then all the non-deterministic refinements are guaranteed to also satisfy those performance con-

straints. This avoids many of the state space explosion issues associated with the verification of non-deterministic systems [46].

Within the scenario approach, the SADF model is used to define such an abstraction of the actual behavior and concrete run-time situations of the system. Moreover, the explicit goal is to define the scenarios such that the abstraction is not overly pessimistic compared to the run-time situations that it represents.

3 Modeling and Analysis of Scenario-Aware Dataflow

3.1 The Scenario-Aware Dataflow Model

This section gives a detailed definition of the scenario-aware dataflow model. We present a formal definition in which we abstract from the streams of data values that are exchanged in the models as well as from the functions that the components compute. We make this abstraction, because we primarily deal with the timing, performance, and resource usage of the models to act as models of scenarios, for which the data values are irrelevant. We do not deal with the functionality that the dataflow graphs realize. An SADF graph consists of a finite set S of scenarios, a mapping $\sigma : S \to \mathscr{G}$ that maps each scenario to a static dataflow graph, i.e., an SDF graph, for that scenario ($\mathscr{G}$ denotes the set of all static dataflow graphs), and a language $\mathscr{L} \subseteq S^{\omega}$ that defines all the possible, infinitely long, scenario sequences. We consider infinitely long scenarios sequences, because many applications we consider are stream processing applications or continuous production machines. The model and theory can, however, be similarly applied to finite scenario sequences. In the WLAN example, $S = \{\mathsf{sync}, \mathsf{header}, \mathsf{payload}, \mathsf{ack}\}$. The definition of the language of scenario sequences for the WLAN application is discussed later.

A static dataflow graph $\sigma(s)$ of a scenario $s \in S$ consists of a set A of *actors* and a set $D \subseteq A \times A$ of *dependencies* between these actors. (For simplicity we assume that at most one dependency exists between any pair of actors, although such a restriction is not necessary, for instance, by defining D as a multiset.) Actors have an *execution time*, given by the function $\tau : A \to \mathbb{R}^{\geq 0}$. A dependency $(a_1, a_2) \in D$ expresses that the firings of actor a_2 depend on the firings of actor a_1. The graph associates with a function $i : D \to \mathbb{N}$ to every dependency $d \in D$ a number, $i(d)$, that determines the precise dependency of firings as follows. In a so-called single-rate graph, it enforces a dependency of firing $n + i(a_1, a_2)$ of actor a_2 on the completion of firing n of actor a_1 for all $n \in \mathbb{N}$. Operationally, we can think of actor a_1 *producing* tokens (one for each firing in a single-rate graph) that are subsequently *consumed* by firings of actor a_2 (also one for each firing in single-rate graph), with $i(a_1, a_2)$ tokens being initially present. Not all graphs are single-rate graphs, i.e., produce and consume exactly one token with each firing and each dependency. Graphs that are not single-rate are called *multi-rate* and some of their actors produce or consume larger, but fixed, quantities of tokens with each firing. For example, the n tokens consumed by a firing

of actor mc from the actor vld in the H.263 model of Fig. 8.1. We use $\pi(a_1, a_2)$ to denote the rate at which actor a_1 produces tokens on the dependency (a_1, a_2) and $\psi(a_1, a_2)$ to denote the rate at which actor a_2 consumes tokens. In general, firing k of a_2 depends on firing m of a_1 if the ranges $[k \cdot \psi(a_1, a_2), (k+1) \cdot \psi(a_1, a_2) - 1]$ and $[m \cdot \pi(a_1, a_2) + i(a_1, a_2), (m+1) \cdot \pi(a_1, a_2) + i(a_1, a_2) - 1]$ overlap, i.e., if firing m of a_1 produces a token that is consumed by firing k of a_2.

Note that firings of an actor have a logical ordering (when we talk about firing k of some actor a). Most of the time, the logical ordering coincides with the temporal ordering in which the firings occur, but this is not necessarily the case. Firings of the same actor can be concurrent (this is called *auto-concurrency*) and occasionally even out of logical order (some firing k starts before firing m although $k > m$). In models in which firings necessarily occur in the same order, logically and temporally, the token dependencies exchanged between actors can be seen as a FIFO queue and are also often implemented in that way. If firings can be out of order, then a more general implementation is needed, such as a windowed cyclic buffer [4]. Many discussions in literature assume that firings occur in temporal order and that dependencies are FIFOs, but such a restriction is not necessary. When actors can complete their firings out of order and the dependencies are realized in FIFO order, then the functionality of the graph may be compromised.

In a single scenario instance, the actors in the dataflow graph of that scenario fire a fixed number of times. Often, this is a minimal, non-empty collection of actor firings after which the graph returns to its original state in terms of tokens and dependencies. This collection of actor firings is called the *repetition vector* of the graph [25, 38]. This kind of scenario is called a *strongly consistent* scenario [43]. In general, however, an arbitrary collection of firings may be defined, denoted with a function $\rho : A \to \mathbb{N}$ that assigns the number of firings to each actor. This may leave the tokens on different edges in the graph than where they were at the start of the scenario. An SADF graph that uses this is called a *weakly consistent* graph [17], assuming that it is still consistent in the sense that in the long run, no matter which scenario sequence it executes, the graph does not deadlock and the number of tokens that can accumulate on any channel is bounded a priori. In general, we will associate a scenario with an SDF graph and a corresponding repetition vector.

For performance analysis purposes, a *reward* specification $r : S \to \mathbb{R}$ may be additionally defined that associates with every scenario a real-valued quantity that captures the amount of progress that is made by that scenario. For example, the throughput requirement for the WLAN application is to process one OFDM symbol every 4μs, but only the scenarios sync, header, and payload process one OFDM symbol. The ack scenario does not. This is captured by assigning a reward of 1 to the former scenarios and a reward of 0 to the latter.

Finally, an SADF model needs to define dependency relations across scenarios, for instance, in the WLAN model the processing of a symbol updates the channel estimation and synchronization parameters. Those results are inputs to the processing of the following scenario. Tokens are used in the SADF model to represent such dependencies. This is done by defining for every scenario a set of *initial tokens*, tokens that are present in the graph in its initial state, that carry dependencies from

earlier scenarios, and *final tokens*, tokens left in the graph at the completion of the scenario, that can carry over dependencies to the following scenarios.

3.2 The Semantics of Scenario-Aware Dataflow

The behavior of an SADF graph is non-deterministic in terms of the sequence of scenarios from the language $\mathscr{L}$ that it executes. This is in fact the only non-deterministic element in the model, the scenario behaviors themselves are deterministic. This non-deterministic behavior of the model can be due to, for instance, data-dependent behavior in the actual system that the model abstracts from. For instance, the number of macro-blocks in a video frame in the H.263 decoder is encoded in the video bit stream, but the model abstracts from the values in that stream. This abstraction is discussed in more detail in Sect. 4.

Figure 8.5 shows the behavior of the WLAN receiver that corresponds to a scenario sequence starting with sync · sync · header · payload · payload · payload · ack in the form of a Gantt chart. Different colors have been used in the figure to represent different scenarios. The rows in the chart correspond to actors and show the firings of those actors. Actors start their firings as soon as all dependencies are satisfied and the firings take an amount of time that is fixed per scenario, but may differ between scenarios. Some actors may fire only in some scenarios. In the example, actors crc and ack fire only in the ack scenario (colored blue).

An important observation to make is that the executions of the scenarios overlap in time. This happens because the different dependencies that carry over from one scenario to the next are satisfied at different points in time and the actor firings in the upcoming scenario start as soon as possible. This is an important advantage of the SADF model, because despite this pipelined execution, it can deal with scenarios in isolation in a compositional manner for all analysis and synthesis purposes.

The view of the behavior of an SADF presented above is called its *operational semantics* and it is the most intuitive view to understand how the model operates. However, it is not the most convenient for mathematical analysis. To understand the mathematical properties of the model, from a temporal point of view, we observe that actor firings wait until all of their dependencies are satisfied, after which they fire for a constant duration. The completion of the firing, in turn, satisfies new dependencies. The time of enabling of the firing can be computed as the maximum of the times at which the individual dependencies are satisfied and the completion time of the firing is computed by adding the execution time to that. We see that the equations that determine how fast the graph executes are constructed from the mathematical operators max and $+$. A lot is known about the algebra that emerges from these two operators [2, 21], which is called max-plus algebra. In particular, and very importantly, it is known to be a linear algebra and it enjoys many of the properties of common linear algebra and has been extensively studied in literature [2, 7, 11, 21].

One of the properties of linear algebra that we use is the fact that a linear system has a canonical representation as a matrix that computes the output and/or next state from the inputs and/or starting state. In the case of a static dataflow graph we consider the initial *state* of the graph as the time stamps (sometimes called *daters*) of the initial tokens in the graph, the times at which the initial dependencies are satisfied. Some graphs have open inputs consuming tokens from the environment. In that case the time stamps of those tokens (not their values!) are considered the inputs of the linear system. After completion of the collection of firings we model by the matrix, the time stamps of the final tokens in the graph represent the next state and any tokens produced on open outputs are the outputs of the system. This leads to an equation of the following form [16, 35]:

$$\begin{bmatrix} \mathbf{x}[k+1] \\ \mathbf{y}[k] \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}[k] \\ \mathbf{u}[k] \end{bmatrix} \tag{8.1}$$

In this equation, the vector $\mathbf{u}$ represents the inputs, vector $\mathbf{x}[k]$ the current state, vector $\mathbf{y}$ the outputs, and vector $\mathbf{x}[k+1]$ the next state. $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ are appropriately chosen matrices that precisely characterize the temporal behavior of the scenario. Note that the matrix–vector multiplication in this equation is in max-plus algebra, not classical linear algebra.

In case the model is closed, i.e., if it does not have inputs or outputs, then the following simple equation remains:

$$\mathbf{x}[k+1] = \mathbf{A}\mathbf{x}[k]$$

In an SADF graph, every scenario s can be individually characterized by a set of matrices $\mathbf{A}_s$, $\mathbf{B}_s$, $\mathbf{C}_s$, and $\mathbf{D}_s$. As an execution follows a particular scenario sequence from the language $\mathscr{L}$, the behavior is determined by a sequence of multiplications with matrices corresponding to the scenarios. In linear systems terminology, an SADF graph is a *switched linear system* in the max-plus linear algebra [44].

We exemplify the max-plus representation using the example SADF of Fig. 8.6. The scenarios are shown in the rightmost part of the figure. We initially assume that the dashed arrows, representing open input and output channels, are not present. The graph has two scenarios: a and b. Each scenario graph consists of three actors: A, B,

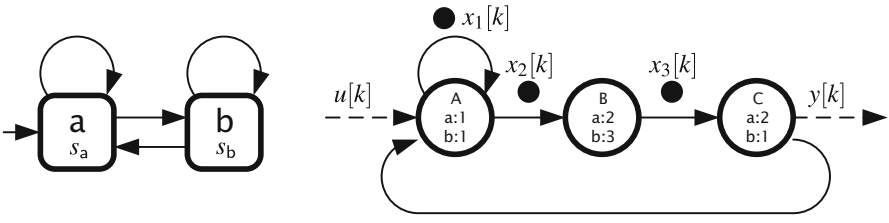


Fig. 8.6 Example SADF graph

and C. Operationally, a scenario consists of a single firing of each of the actors. The difference between the scenarios is only in the firing delays of actors B and C. In particular, in scenario a the firing of actors B and C will take 2 time units each, while in scenario b the firings will take 3 and 1 time units, respectively. Consequently, the scenario matrices differ. The matrices can be computed as explained in [15]. Here, we present the outcomes:

$$\mathbf{A}_{\mathsf{a}} = \begin{bmatrix} 1 & -\infty & 3 \\ 1 & -\infty & 3 \\ -\infty & 2 & -\infty \end{bmatrix} \quad \text{and} \quad \mathbf{A}_{\mathsf{b}} = \begin{bmatrix} 1 & -\infty & 2 \\ 1 & -\infty & 2 \\ -\infty & 3 & -\infty \end{bmatrix}$$

In the matrices, entry $[\mathbf{A}]_{i,j}$ specifies the time distance between the time stamp of initial token j and the time stamp of the final token i. In our example, the initial tokens coincide with the final tokens with token indices increasing from left to right w.r.t. the rightmost graph in Fig. 8.6. An entry $-\infty$ indicates that the corresponding tokens are independent.

We now turn our attention to the dashed input and output channels of the scenario graphs of Fig. 8.6. Their behavior is included by also specifying the matrices **B**, **C**, and **D** of Eq. 8.1. Matrix **B** captures the dependency of the final internal state on the input token. In this case it is a column vector as there is only one input token consumed. It is identical for both scenarios and has the value:

$$\mathbf{B}_{\mathsf{a}} = \mathbf{B}_{\mathsf{b}} = \begin{bmatrix} 1 \\ 1 \\ -\infty \end{bmatrix}$$

Only the left two tokens, x_1 and x_2, depend on the input and the timing distance is the execution time of actor A. Matrix **C** represents the dependency of the output token on the tokens of the initial internal state. In this case it is a row vector with the following values for the scenarios:

$$\mathbf{C}_{\mathsf{a}} = \begin{bmatrix} -\infty & -\infty & 2 \end{bmatrix} \quad \text{and} \quad \mathbf{C}_{\mathsf{b}} = \begin{bmatrix} -\infty & -\infty & 1 \end{bmatrix}$$

In both cases the output only depends on the rightmost token, x_3, and the timing distance is the execution time of actor C. The final part is the matrix **D**, which in this case reduces to a scalar, because there is only one input and one output, with the trivial value of $-\infty$, because there is no direct dependency from the input to the output.

$$\mathbf{D}_{\mathsf{a}} = \mathbf{D}_{\mathsf{b}} = -\infty$$

The characteristic matrices completely define the timing of the scenarios. The behavior of the SADF consists of sequences of scenarios in the language $\mathscr{L}$. The language of scenario sequences can be specified with formalisms to define

languages over a finite alphabet (in this case the set of scenarios). Well-known examples are finite state automata (FSAs) and regular expressions (both are in fact equally expressive). In the example SADF of Fig. 8.6, the scenario sequences are specified by the FSA on the left. It defines the language of all sequences of scenarios a and b that start with the scenario a. In [1], the language of scenario sequences is expressed with regular expressions including an explicit repetition construct, leading to a compact representation with an efficient analysis.

One can optionally include information about the likelihood of the occurrence of sequences of scenarios. For instance, by adding probabilities to the finite state machine, resulting in a Markov Chain representation that defines a σ-algebra on the language $\mathscr{L}$. This gives the model well-defined notions of stochastic behavior, such as *expected* throughput or *variance* in latency [43].

The semantics we have introduced can be used to define the explicit *state space* of an SADF graph in which the scenario sequences are defined by an FSA. The *state* of an SADF can be captured by a combination of the current state of the FSA and the current time-stamp vector of the tokens. For example, the initial state of the SADF graph in Fig. 8.6 is the pair $\left(s_{\mathsf{a}}, \left[0\ 0\ 0\right]^T\right)$. After executing the scenario a and the FSA non-deterministically moving to state s_{b}, the state would become $\left(s_{\mathsf{b}}, \left[3\ 3\ 2\right]^T\right)$. Note that in many discrete as well as continuous and hybrid models, the state of a state space refers to a snapshot of the integral system at some point in the (physical or modeling) time domain. For SADF, and max-plus models in general, this is not the case. It refers, instead, to the state before or after the execution of certain scenarios, where the elements of the time-stamp vector of such a state refer to possibly different time stamps in the time domain.

Every scenario execution leads to a discrete transition in the state space as the FSA moves to a new state and the time-stamp vector changes accordingly. Naturally such a state space would be infinite, as the time stamps increase and diverge towards infinity. Such an infinite state space could not be constructed in practice and could not be used for any analysis. We therefore apply a normalization strategy to keep the state space finite. This strategy is based on the observation that the scale of a time-stamp vector has no impact on the possible future behaviors of the SADF in a given state. If an SADF, from a given state $(s, \mathbf{x})$, can perform a sequence of scenarios leading to a state $(s', \mathbf{x}')$, then the state $(s, \mathbf{x} + c)$ can perform the same sequence of scenarios leading to the state $(s', \mathbf{x}' + c)$ for any $c \in \mathbb{R}$. Therefore, we only explicitly record a state using its normalized time-stamp vector, i.e., as $\mathbf{x} - c$ for $c \in \mathbb{R}$ such that $|\mathbf{x} - c| = 0$. Hence, the state $\left(s_{\mathsf{b}}, \left[3\ 3\ 2\right]^T\right)$ is recorded as $\left(s_{\mathsf{b}}, \left[0\ 0\ -1\right]^T\right)$. This way the relative differences of the time stamps are recorded, but not their absolute values. To be able to account for the amount of time passing in the transition from the initial state to this state, the normalization constant c (in the example c is equal to 3) is associated with the state transition. If we follow this approach for the example of Fig. 8.6, we arrive at the finite state space that is shown in Fig. 8.7 (the bold and dashed arrows and circles are explained in the following section). The transitions in the state space are additionally decorated with the scenario that is executed.

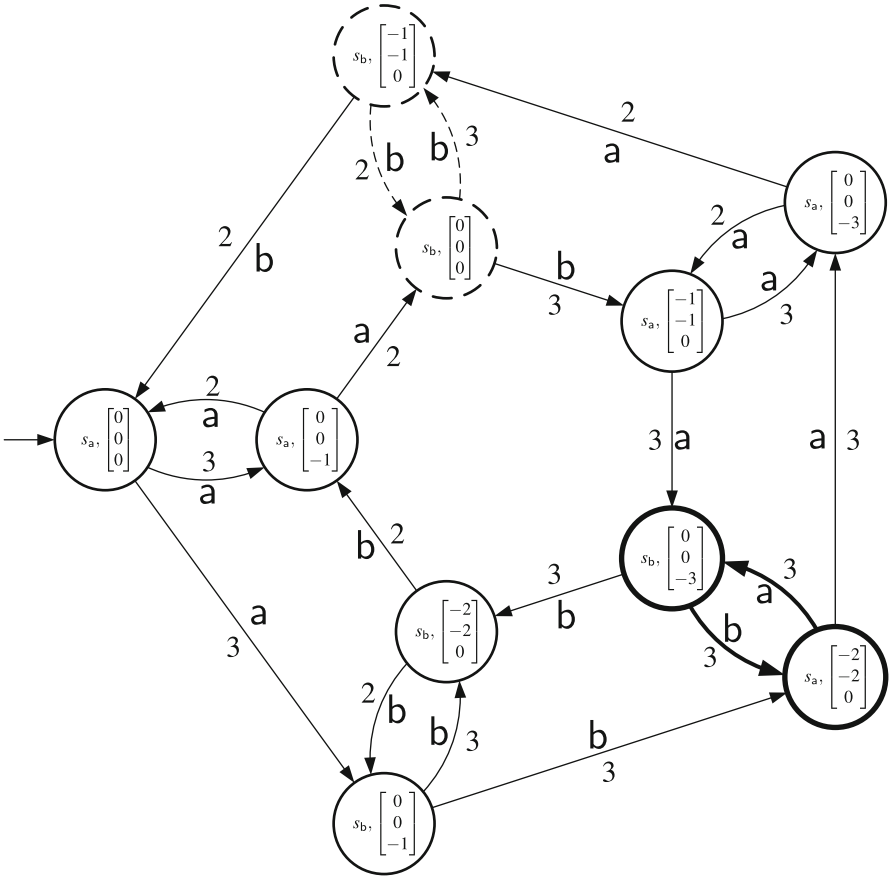


Fig. 8.7 State space of the SADF of Fig. 8.6

The precise definition of the state space of an SADF is as follows. The set Σ of states of the state space are pairs:

$$\Sigma = \{(q, \mathbf{x}) \in Q \times (\mathbb{R} \cup \{-\infty\})^n \mid |\mathbf{x}| = 0\}$$

where Q is the set of states of the FSA and n is the size of the state vector. The transitions $((q_1, \mathbf{x}_1), d, (q_2, \mathbf{x}_2))$ of the state space are triples from $\Sigma \times \mathbb{R} \times \Sigma$, such that (i) there is a transition from state q_1 labeled with scenario s, to state q_2 in the FSA, (ii) $\mathbf{x}_2 + d = \mathbf{A}_s \mathbf{x}_1$. We usually refer to the state space as only the set of all states that are reachable from the initial state $(q_0, \mathbf{0})$, where q_0 is the initial state of the FSA, and the corresponding transitions. Moreover, for convenience, we may additionally label the transitions in the state space with the scenarios that they correspond to and/or with the rewards corresponding to these scenarios, as we have done in the state space in Fig. 8.7.

The state space allows us to determine the state vector obtained after any finite scenario sequence that is a prefix of a word in the scenario language as follows. We follow a path labeled with the scenarios in the scenario sequence through the state space, starting from the initial state. If the sum of the edge weights on the path is d and the normalized time stamp vector of the final state is $\mathbf{x}$, then the final state vector after the scenario sequence is equal to $\mathbf{x} + d$.

3.3 *Performance Analysis of Scenario-Aware Dataflow*

If the language of scenario sequences is a regular language, like in the example of Fig. 8.6, and defined by an automaton without acceptance conditions, then the structure is called a *max-plus automaton* [12]. There are known techniques to compute from such a max-plus automaton the worst-case number of scenarios per time unit [12], or, if the scenarios are annotated with rewards, the worst-case total reward per time unit [15, 17]. Those methods can also report the critical scenario sequence and the critical path of actor firings within the scenarios. If the automaton does have acceptance conditions, then the same analysis can be applied, also with exact results, by identifying and subsequently removing any states that can only occur a finite number of times in any accepted word, using standard automata analysis techniques.

The worst-case throughput analysis centers around the structure called max-plus automaton graph (MPAG). In particular, given an SADF graph, the structure is constructed as follows: a vertex is created for each initial token of a scenario of an FSA state. If $[\mathbf{A}_{\mathsf{s}}]_{i,j} \neq -\infty$ and there is a transition in the FSA from state m labeled with scenario r to a state n labeled with scenario s, an edge is created from node j in state m to node j in state n with weight $[\mathbf{A}_{\mathsf{s}}]_{i,j}$. The MPAG of the SADF of Fig. 8.6 is shown in Fig. 8.8. Maximum cycle mean (MCM) [10] analysis of the MPAG structure will identify the critical scenario sequence, i.e., the sequence with the worst-case average amount of time taken per scenario. The inverse of the maximum cycle mean (MCM) equals the worst-case throughput of the graph, the worst-case number of scenarios executed per time unit. For the running example of Fig. 8.6, the critical scenario sequence is captured with bold arrows in Fig. 8.8.

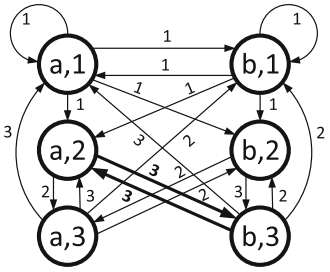


Fig. 8.8 Max-plus automaton graph of the SADF of Fig. 8.6

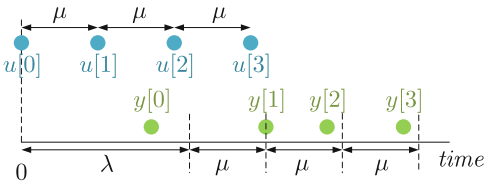


Fig. 8.9 Definition of latency

This is the sequence $(\mathsf{ab})^\omega$. The corresponding MCM is equal to 3 defining the greatest lower bound on throughput of any scenario sequence, which is equal to $1/3$ scenarios per time unit. Note that in practice, a scenario often represents a coherent set of computations, like decoding one audio/video frame. As explained earlier, in some SADF models, the amount of progress differs per scenario, as in the WLAN example. In such cases rewards are used to specify the progress. The MPAG is then extended by annotating the edges additionally with the reward of the corresponding scenario and a maximum cycle ratio (MCR) analysis is performed in place of the MCM analysis [17]. The MCR gives us the worst-case (minimum) amount of reward per time unit.

The linear model also facilitates the computation of latency. Latency is often defined with respect to a periodic input that delivers inputs to the system starting at time 0 and with some period $\mu > 0$. This is illustrated in Fig. 8.9. We assume a single input $u[k]$ and a single output $y[k]$, but the definition and the analysis are easily generalized to multiple inputs and outputs. Latency is defined as the smallest value $\lambda \in \mathbb{R}$ such that $y[k] \leq \lambda + k \cdot \mu$ for all $k \in \mathbb{N}$. For the example of Fig. 8.9, for any smaller value of λ than indicated, $y[1] > \lambda + 1 \cdot \mu$. If such a value of λ is found, then the system is a refinement, in the sense of Sect. 2.2, of a system that produces outputs periodically with period μ after an initial delay of λ.

Note that there may be a trade-off between throughput and latency. For a higher throughput (smaller value of the period μ), the latency may be larger. Their relation is always monotone. Moreover, if $\frac{1}{\mu}$ is higher than the maximal throughput of the graph, then such a value λ does not exist and the graph does not have a latency.

For multiple outputs, the definition is generalized as follows. The latency is the smallest vector $\boldsymbol{\lambda}$ such that

$$\mathbf{y}[k] \leq \boldsymbol{\lambda} + k\mu \quad \text{forall} \quad k \in \mathbb{N}$$

This vector can be computed as:

$$\boldsymbol{\lambda} = \max_{k \in \mathbb{N}} \mathbf{y}[k] - k\mu$$

In max-plus algebra it is common to use $\oplus$ as short-hand notation for the binary max operator and $\bigoplus$ for the max quantifier. In this notation we get the following equation:

$$\boldsymbol{\lambda} = \bigoplus_{k \in \mathbb{N}} \mathbf{y}[k] - k\mu$$

Following straightforward max-plus linear algebra computations we can derive that the latency for a static dataflow graph with one scenario with matrices **A**, **B**, **C**, and **D** is computed as follows [1]:

$$\boldsymbol{\lambda} = \mathbf{C}\,(\mathbf{A} - \mu)^*\,(\mathbf{x}[0] \oplus (\mathbf{B0} - \mu)) \oplus \mathbf{D0} \tag{8.2}$$

where the $*$-closure of a square matrix **M** is defined as

$$\mathbf{M}^* = \bigoplus_{k=0}^{\infty} \mathbf{M}^k$$

Note that this closure exists in the latency computation of Eq. 8.2 if and only if the graph has a latency. It can be efficiently computed exactly. We observe further that the latency also depends on the initial state, $\mathbf{x}[0]$ of the system.

We next show that the derivation of the latency of a max-plus linear system is a straightforward exercise. The following equation can be shown to hold for the state vector $\mathbf{x}[k]$ by induction on k and the fact that $\mathbf{x}[k+1] = \mathbf{A}\mathbf{x}[k] \oplus \mathbf{B}\mathbf{u}[k]$.

$$\mathbf{x}[k] = \mathbf{A}^k\mathbf{x}[0] \oplus \left(\bigoplus_{m=0}^{k-1} \mathbf{A}^m\mathbf{B}\mathbf{u}[k-m-1]\right)$$

Then the output vector can be computed as

$$\begin{aligned}\mathbf{y}[k] &= \mathbf{C}\mathbf{x}[k] \oplus \mathbf{D}\mathbf{u}[k] \\ &= \mathbf{C}\left(\mathbf{A}^k\mathbf{x}[0] \oplus \bigoplus_{m=0}^{k-1} \mathbf{A}^m\mathbf{B}\mathbf{u}[k-m-1]\right) \oplus \mathbf{D}\mathbf{u}[k]\end{aligned}$$

From this the latency can be computed using the periodic inputs $\mathbf{u}[k] = k \cdot \mu \cdot \mathbf{0}$

$$\begin{aligned}\boldsymbol{\lambda} &= \bigoplus_{k\in\mathbb{N}} \mathbf{y}[k] - k\cdot\mu \\ &= \bigoplus_{k\in\mathbb{N}} \mathbf{C}\left(\mathbf{A}^k\mathbf{x}[0] \oplus \bigoplus_{m=0}^{k-1} \mathbf{A}^m\mathbf{B}\mathbf{u}[k-m-1]\right) \oplus \mathbf{D}\mathbf{u}[k] - k\cdot\mu \\ &= \mathbf{C}\bigoplus_{k\in\mathbb{N}} (\mathbf{A} - \mu\mathbf{I})^k\,\mathbf{x}[0] \oplus \bigoplus_{k\in\mathbb{N}} \left(\mathbf{C}\bigoplus_{m=0}^{k-1} \left(\mathbf{A}^m\mathbf{B}\cdot(k-m-1)\cdot\mu\cdot\mathbf{0}\right) \oplus \mathbf{D}\cdot k\cdot\mu\cdot\mathbf{0}\right) \\ &\quad - k\cdot\mu \\ &= \mathbf{C}(\mathbf{A} - \mu)^*\,\mathbf{x}[0] \oplus \bigoplus_{k\in\mathbb{N}} \left(\mathbf{C}\bigoplus_{m=0}^{k-1} \mathbf{A}^m\mathbf{B}\cdot(k-m-1)\cdot\mu\cdot\mathbf{0} - k\mu\right) \oplus \bigoplus_{k\in\mathbb{N}} \mathbf{D}\cdot k\cdot\mu\cdot\mathbf{0} \\ &\quad - k\cdot\mu\end{aligned}$$

$$
\begin{aligned}
&= \mathbf{C}\,(\mathbf{A}-\mu)^*\,\mathbf{x}[0] \oplus \mathbf{C} \bigoplus_{k\in\mathbb{N}} \left(\bigoplus_{m=0}^{k-1} (\mathbf{A}-\mu)^m \right) \mathbf{B} \cdot (-1) \cdot \mu \cdot \mathbf{0} \oplus \mathbf{D0} \\
&= \mathbf{C}\,(\mathbf{A}-\mu)^*\,\mathbf{x}[0] \oplus \mathbf{C} \left(\bigoplus_{k\in\mathbb{N}} \bigoplus_{m=0}^{k-1} (\mathbf{A}-\mu)^m \right) (\mathbf{B0}-\mu) \oplus \mathbf{D0} \\
&= \mathbf{C}\,(\mathbf{A}-\mu)^*\,\mathbf{x}[0] \oplus \mathbf{C}\left(\mathbf{A}-\mu^*\right)(\mathbf{B0}-\mu) \oplus \mathbf{D0} \\
&= \mathbf{C}\,(\mathbf{A}-\mu)^*\,(\mathbf{x}[0] \oplus (\mathbf{B0}-\mu)) \oplus \mathbf{D0}
\end{aligned}
$$

Note that although the definition is given for a periodic input, the definition is not restricted to periodic sources and linearity allows it to be used to predict the latency of other types of sources as well. A more elaborate discussion is given by Moreira [28].

The latency computation can also be generalized to switching scenarios in SADF, where given an SADF and an automaton defining the possible scenario sequences, we can compute the exact worst-case latency. Details are beyond the scope of this chapter and can be found in [1]. Acceptance conditions on the automaton can be additionally taken into account by identifying and excluding from the analysis any states of the automaton that cannot be reached in any accepting word.

Throughput analysis is also possible on the state space of an SADF. The worst-case throughput can be determined from the worst-case, the maximum cycle mean, of the delay values on the edges, or the maximum cycle ratio when rewards are considered. In the state space of Fig. 8.7, this is the cycle indicated in bold, corresponding to the alternating execution of scenarios a and b. Note that it has the same cycle mean (3) as the MPAG in Fig. 8.8. Typically, one prefers the analysis on the MPAG though, because the MPAG is usually, as in this example, smaller than the state space. The state space does have the advantage that one can also determine the best-case throughput, in the sense of the highest throughput one could guarantee under the assumption that one has full control over the sequence of scenarios that occur. This may be the case, for instance, when the scenarios represent decisions made by a scheduler as in the work of Yang et al. [49], or a supervisory controller as the work of Van der Sanden et al. [45, 46]; see also Sect. 4.3 in Chap. 9. This is achieved by finding a cycle in the state space with a *minimum* cycle mean (or ratio). One of the possible optimal scenario sequences is indicated in Fig. 8.7 with dashed arrows and circles. Note that such an optimal-throughput scenario sequence cannot be determined from the MPAG. In particular, the minimum cycle mean in the MPAG equals 1, which does not correspond to the optimal-throughput cycle in the state space, which has a cycle mean of $5/2$. This can be explained as follows. The MPAG represents all dependencies between individual tokens across different scenarios. The worst-case cycle in this graph simultaneously identifies the worst-case scenario sequence and the worst-case among all token dependencies. The best-case scenario sequence in the state space still needs to consider the worst-case token dependencies

for its throughput. This combination of best-case scenarios and worst-case token dependencies cannot be identified in any cycles in the MPAG.

The optimal-throughput solution can also be generalized to the situation that the transitions on the FSA can be partitioned into transitions that are controllable and transitions that are uncontrollable. In that case, optimal attainable throughput can be analyzed by finding optimal strategies in a two-player cycle mean game (or a cycle ratio game in case of rewards) [45].

3.4 Modeling Switched Max-Plus Linear Systems

The scenario-aware dataflow model is an instance of a *switched linear system* in the max-plus linear algebra, i.e., a system that switches between different linear behaviors. Many other models fall in the same category and similar techniques can be applied for their performance analysis. An example is the activity model introduced in Chap. 9. To fit with the switched linear model, the systems need to adhere to linear modes of operation.

Example 1 Consider a conveyor belt of length l that moves at a constant speed v as shown in Fig. 8.10a. It can transport objects. For simplicity we ignore the physical dimensions of the objects, so they can be arbitrarily close on the belt and can be placed on the belt at any time. We model the belt as a system B that takes a (possibly infinite) sequence of input events $u[k]$, which are the points in time at which object k is placed on the start of the belt. The outputs of the system B, $y[k]$, are the time points at which object number k reaches the end of the belt. It satisfies the following equation:

$$y[k] = u[k] + \frac{l}{v}$$

For a system to be linear, it needs to have two properties. It should be *additive* and it should be *homogeneous*. Both are to be interpreted in terms of max-plus algebra.

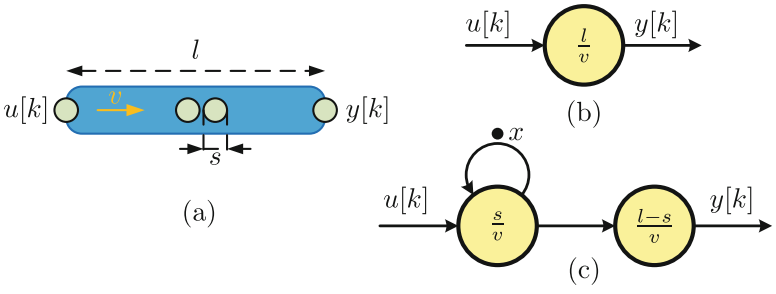


Fig. 8.10 Example of a conveyor belt and its dataflow linear model

We assume a system S with a scalar input and a scalar output. The notation and definitions generalize straightforwardly to systems with multiple inputs and outputs and to sequences of inputs or outputs. We use $S(x)$ to denote the time stamp of the output event of system S in response to an input event with time stamp x. Max-plus additivity means that a system satisfies the following rule for all x_1 and x_2:

$$S(\max(x_1, x_2)) = \max(S(x_1), S(x_2))$$

A more intuitive term for max-plus additivity is *monotonicity* as this is equivalent to the following condition:

$$x_1 \leq x_2 \Rightarrow S(x_1) \leq S(x_2)$$

An important consequence of additivity / monotonicity is that it allows the superposition principle to be applied [13]. This principle states that the response due to the sum of a number of inputs (in our setting the maximum of inputs) can be determined as the sum of the outputs that are due to each of the inputs individually. An example of the principle is given later in this section.

It is easy to see that the belt of Example 1 is additive.

$$B(\max(x_1, x_2)) = \max(x_1, x_2) + \frac{l}{v} = \max(x_1 + \frac{l}{v}, x_2 + \frac{l}{v}) = \max(B(x_1), B(x_2))$$

The second property required for a system to be max-plus linear is homogeneity. For a max-plus linear system this means that the following must hold for all x and c.

$$S(c + x) = c + S(x)$$

A more intuitive term for max-plus homogeneity is *shift-invariance*, if the time stamp of the input event is shifted by an amount c, then the time stamp of the output event is shifted by the same amount.

The belt is also easily seen to be shift-invariant.

$$B(c + x) = (c + x) + \frac{l}{v} = c + (x + \frac{l}{v}) = c + B(x)$$

Hence, the belt is max-plus linear. It has no internal state, so its canonical representation consists of only the $\mathbf{D}$ matrix, which, because there is only one input and only one output, is just a scalar number with the value $\mathbf{D} = l/v$. The belt also has a, very simple, dataflow representation as a single actor, as shown in Fig. 8.10b. Figure 8.10c additionally shows a dataflow model of a belt where we do take the physical size of objects into account. We assume it is s. Now the system has internal state x to remember how much space/time the previous object takes. The canonical representation is as follows. Its derivation is left as an exercise for the reader.

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} = \begin{bmatrix} s/v & s/v \\ l/v & l/v \end{bmatrix}$$

For this system with internal state it is interesting to consider using the superposition principle to determine the output sequence that is the response to the input sequence with $u[0] = 4$ and $u[1] = 5$. We assume that $l/v = 10$ and $s/v = 2$. The input sequence u can be seen as $u = u_0 \oplus u_1$, where u_0 considers only the first input, i.e., $u_0[0] = 4$ and $u_0[1] = -\infty$, and u_1 considers only the second input, i.e., $u_1[0] = -\infty$ and $u_1[1] = 5$. The superposition principle predicts that if y is the output corresponding to u and y_0 and y_1 are the outputs corresponding to u_0 and u_1, respectively, then $y = y_0 \oplus y_1$. Assuming $x[0] = 0$, we have according to Eq. 8.1:

$$\begin{bmatrix} x[1] \\ y_0[0] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} x[0] \\ u_0[0] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 \\ 14 \end{bmatrix}$$

$$\begin{bmatrix} x[2] \\ y_0[1] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} x[1] \\ u_0[1] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} 6 \\ -\infty \end{bmatrix} = \begin{bmatrix} 8 \\ 16 \end{bmatrix}$$

Hence, $y_0[0] = 14$ and $y_0[1] = 16$. In the same way we can compute y_1:

$$\begin{bmatrix} x[1] \\ y_1[0] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} x[0] \\ u_1[0] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} 0 \\ -\infty \end{bmatrix} = \begin{bmatrix} 2 \\ 10 \end{bmatrix}$$

$$\begin{bmatrix} x[2] \\ y_1[1] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} x[1] \\ u_1[1] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} 2 \\ 5 \end{bmatrix} = \begin{bmatrix} 7 \\ 15 \end{bmatrix}$$

Therefore, $y_1[0] = 10$ and $y_1[1] = 15$. We combine both results to conclude that $y[0] = y_0[0] \oplus y_1[0] = 14$ and $y[1] = y_0[1] \oplus y_1[1] = 16$. This can easily be verified to be the correct result in which the first input takes exactly 10 time units (the length of the belt, l/v) from input to output, but the second input on the belt is delayed by the first, because they need to be two time units (s/v) apart. Despite the fact that the two inputs "interact," their responses can be computed individually and then combined.

Note that the system implementation does not necessarily need to be linear, it only needs to have a linear abstraction in terms of the abstraction relation discussed in Sect. 2.2. An example is the response time of a job scheduled on a processor. If the arrival time of the job is the input event and its completion time is the output event, then the scheduler is often not shift-invariant. It may still have a linear abstraction. For instance, when the scheduler has a worst-case response time for the job. In that case, the model that adds the worst-case response time to the job arrival time is a linear abstraction of the job scheduled on the processor.

3.5 Parametric Analysis

For the worst-case throughput analysis technique explained above to be applicable, the actor firing delays and rates characterizing SDF scenarios must be *fixed* and *known at design-time*. If we deem actor firing delays and rates as *parameters*, then assigning values to those parameters yields an SADF that is amenable to the analysis described.

If we have a system with a number of possible values for each of the parameters, to perform the analysis as described previously in this chapter, we will consequently need to generate a scenario for every possible combination of assignments. Hence, the number of scenarios may get to a point where it will experience compactness [5] or succinctness-related [42] problems. On the analysis side, the product set cardinality hampers the use of SADF in the analysis of systems exposing high levels of data-dependent dynamics in a way that it will render the analysis run-time prohibitive because of the shear number of scenarios that need to be considered. The problem becomes even more intricate if parameters are dependent and not all combinations of values can occur. The dependencies may be specified explicitly in the construction of the model, e.g., one parameter is given as an expression of another or implicitly, e.g., a dataflow scheduler synthesizes some parameter values [31].

We can address these problems by combining the finite control of SADF with parameterized dataflow into a construct we refer to as parameterized SADF. In particular, we model each scenario using a parameterized dataflow graph.

Parameterization, as a syntactic construct, allows us to represent vast sets of scenarios in a compact way (parameters help keep the size of the model manageable) and to explicitly represent the dependencies between parameters as constraints. The underlying parametric analysis enables us to avoid the enumeration of the parameter product set. An example of a parameterized SADF graph is shown in Fig. 8.11. The dataflow graph on the right-hand side of the figure reveals three parameters: p, q, and r. Parameters p and q attain values from the set of non-negative integers and are

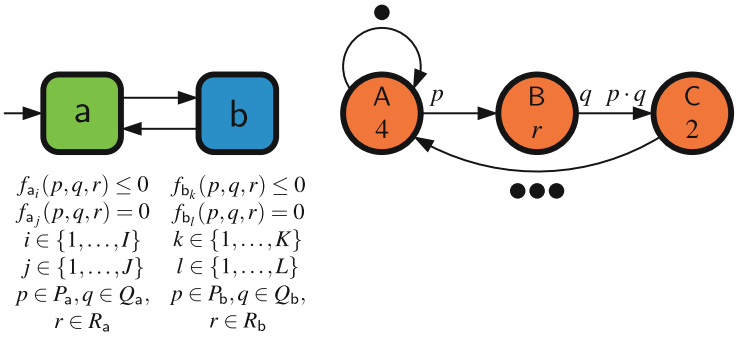


Fig. 8.11 Example of a parameterized SADF

used to parameterize rates of actors A, B, and C, while parameter r attains values from the set of non-negative real numbers and is used to parameterize the firing delay of actor B. The FSA in the left part of the figure indicates that the structure involves two scenarios a and b. Parameter dependencies and parameter bounds are specified per scenario in terms of *scenario domains* described (in the most general case) via:

- a system of non-linear inequalities

$$f_{\mathsf{a}i}(p, q, r) \leq 0$$

where $i \in \{1, \ldots, I\}$, $p \in P_{\mathsf{a}} \subset \mathbb{N}_0$, $q \in Q_{\mathsf{a}} \subset \mathbb{N}_0$, $r \in R_{\mathsf{a}} \subset \mathbb{R}_{\geq 0}$, and

$$f_{\mathsf{b}k}(p, q, r) \leq 0$$

where $k \in \{1, \ldots, K\}$, $p \in P_{\mathsf{b}} \subset \mathbb{N}_0$, $q \in Q_{\mathsf{b}} \subset \mathbb{N}_0$, $r \in R_{\mathsf{b}} \subset \mathbb{R}_{\geq 0}$

- and a system of non-linear equalities

$$f_{\mathsf{a}j}(p, q, r) = 0$$

where $j \in \{1, \ldots, J\}$, $p \in P_{\mathsf{a}}$, $q \in Q_{\mathsf{a}}$, $r \in R_{\mathsf{a}}$, and

$$f_{\mathsf{b}l}(p, q, r) = 0$$

where $l \in \{1, \ldots, L\}$, $p \in P_{\mathsf{b}}$, $q \in Q_{\mathsf{b}}$, $r \in R_{\mathsf{b}}$.

Each transition of the scenario FSA incurs the invocation of an arbitrary instance of the parameterized scenario that the transition destination state corresponds to. An instance of a parameterized scenario is a concrete scenario obtained by assigning all parameters with values inside the parameterized scenario domain, i.e., values that satisfy the constraints. The operational semantics of the model is illustrated in Fig. 8.12. For illustrative purposes, scenario domains are depicted as *2-D* planes (recall that a domain can be non-linear too) in the $p - q - r$ space. For example, whenever a transition $\mathsf{a} \rightarrow \mathsf{b}$ is taken, a concrete scenario obtained by assigning values from the b scenario domain to parameters p, q, and r is executed. Example of such assignments is depicted by the points $(p_{\mathsf{a}}, q_{\mathsf{a}}, r_{\mathsf{a}})$ and $(p_{\mathsf{b}}, q_{\mathsf{b}}, r_{\mathsf{b}})$.

Throughput analysis for parameterized SADF is based on the max-plus switched linear system semantics of SADF. In particular, starting from a set of parameterized scenarios, for each of them, a single representative max-plus matrix is generated that captures the worst-case system behavior per parameterized scenario. The matrix is derived by solving a series of non-linear optimization problems. Thereafter, the matrices are used, in the same way, to construct the corresponding MPAG the MCM of which defines the inverse of the worst-case throughput. Experimental results indicate that the approach is advantageous for both modeling and analysis perspective for graphs with broad-ranging interdependent parameters. More detailed explanations can be found in [36, 37].

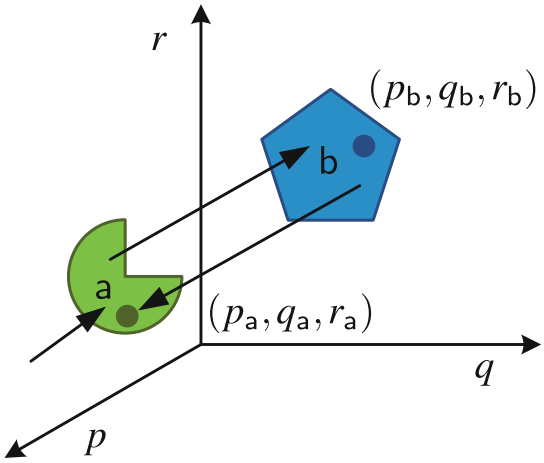


Fig. 8.12 Operational semantics of the parameterized SADF of Fig. 8.11

	Parametric execution time		
Scenario	A_0	A_1	A_2
a	$6p_1$	$2.5p_2$	$2p_2$
b	$1p_1$	$4.5p_2$	$1p_2$

Parameter range
$1 \leq p_1 \leq 5$
$1 \leq p_2 \leq 5$

Fig. 8.13 An SADF with parametric actor firing delays

The concept of parameterized SADF presented above treats the case where parameters are not fixed during the execution of the systems. In particular, parameters are allowed to change from one invocation of a scenario to the next invocation of the same scenario. The analysis results in a greatest lower bound of the performance of any possible scenario sequence and parameter valuations. However, we may often encounter systems where parameters do not change during the system execution, or change only infrequently. For such a parameterized SADF, instead of a single worst-case performance result, we can find throughput expressions that present the throughput as a function of the scenario parameter values (actor firing delays and actor port rates). Calculation of throughput for a particular parameter valuation is then merely an evaluation of this function for the specific parameter values, which is much faster than the standard throughput analysis. This result may be particularly important for run-time management, which we discuss at the end of the chapter.

We first consider the case where actor firing delays can be parameters. This setting was first discussed in [18] for SDF and in [8] for SADF. Consider the example SDF shown in Fig. 8.13. The scenario FSA is shown in the left part of the figure, while the scenario graph is in the right part of the figure. By now a reader will

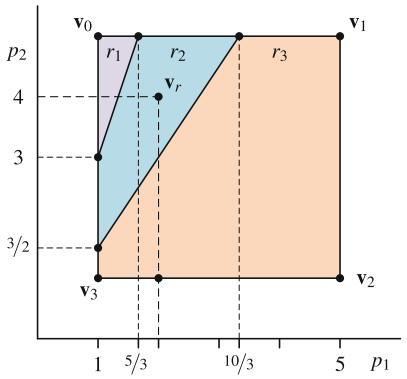


Fig. 8.14 Throughput regions of the parameterized SADF of Fig. 8.13

be able to deduce that the model involves two scenarios: a and b. In each scenario, actors A_0, A_1, and A_2 perform one firing each. The firing delays are functions of two parameters, p_1 and p_2, as specified in the table in the lower part of Fig. 8.13.

When the actor firing delays are given as linear expressions of parameters, it can be shown that parts of the parameter space that share the same MCM expression (recall that the inverse of the MCM defines the throughput) form convex polyhedra. We call these convex polyhedra *throughput regions*.

For the example SDF of Fig. 8.13, the throughput regions are shown in Fig. 8.14. Values of the parameter p_1 span the x-axis, while the values of p_2 span the y-axis. The model has three throughput regions denoted r_1, r_2, and r_3 with the following expressions for the MCM μ:

- $\mu = 5.5p_2$ if $3p_1 \leq p_2$ for $(p_1, p_2) \in r_1$;
- $\mu = 3p_1 + 4.5p_2$ if $p_2 \leq 3p_1 \leq 2p_2$ for $(p_1, p_2) \in r_2$, and
- $\mu = 6p_1 + 2.5p_2$ if $3p_1 \geq 2p_2$ for $(p_1, p_2) \in r_3$.

The inverses of MCM expressions define the throughput expressions. For more details we refer the reader to [8].

The approach we just presented is only applicable to graphs with parameterized actor firing delays. Furthermore, these delays are constrained to be linear combinations of parameters. Indeed, the key assumption in [8] is linearity. Introduction of parameterized rates implies non-linearity as products of rates may appear in MCM expressions. Therefore, the results of [8] were generalized in [34], which can deal with graphs of certain structure involving both parameterized rates and actor firing delays. The technique linearizes the problem by expanding it into a high-dimensional space. Unfortunately, although theoretically relevant, the technique is of limited usability (limited to a set of only a few critical parameters) because manipulation of high-dimensional polytopes incurs a high penalty in performance.

Listing 1 Pseudo-code of an abstract video decoder

```
1: frame = buffer_frame()
2: if detect_frame_type(frame) == full then
3:   x = decode_full(frame)
4:   sub = subtitle_overlay(x)
5: else
6:   x = decode_delta(frame)
7: end if
8: output = construct_frame(x)
9: display(output, sub)
```

4 A Programming Model for SADF

SADF is a dataflow model that can represent scenarios of pipelined applications. It can also be used as a programming paradigm for dynamic pipelined applications in which the dynamic behavior can be expressed as different modes of operation. An additional advantage of such an approach is that the application structure can directly guide the process of design-time scenario identification. This section describes such a programming model based on SADF and its realization on the CompSOC platform [47, 48]. It explicitly addresses the challenge of run-time scenario identification and the required switching and reconfiguration to execute the corresponding behavior.

4.1 A Scenario-Aware Dataflow Application

We consider streaming real-time applications that process data when it is received. The control flow of such applications often depends on the received data. Upon receiving a video frame, for example, a decoder detects if it is a full frame or a delta frame and invokes the appropriate decoding function. In a sequential language, a programmer may solve such a dependency with a simple *if-else* construct as shown in Listing 1. This likely leads to the identification of both cases as separate scenarios.

SADF is a natural way to describe the behavior of this application and captures different input-dependent control flows in its *scenarios*. Every possible control flow is captured in a *scenario graph* by the programmer, possibly based on existing sequential code. The scenario graph for decoding a full video frame (S_{full}) is depicted in Fig. 8.15. When a delta frame is detected, the application behavior and thus also the scenario graph are different, namely S_{delta} in Fig. 8.16. Actor names are abbreviations of the functions in Listing 1. The possible scenario sequences are specified by the FSM in Fig. 8.17.

The SADF model allows tight analysis of applications with input-data dependent control flow, it does not define an *implementation* model. Consider the video decoder, where bf and dft are always executed first in either scenario. Only after

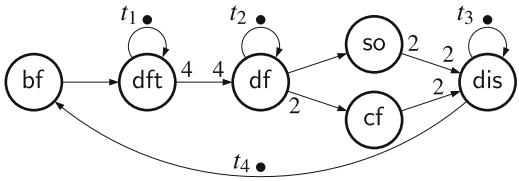


Fig. 8.15 Scenario graph, S_{full}, for decoding a full video frame

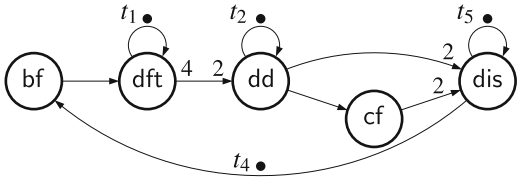


Fig. 8.16 Scenario graph, S_{delta}, for decoding a delta video frame

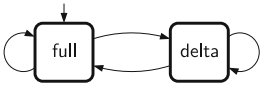


Fig. 8.17 The FSM of the video decoder with scenarios full and delta

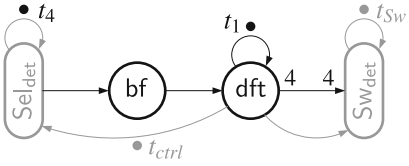


Fig. 8.18 Analysis graph, S_{det}, of the detector scenario det

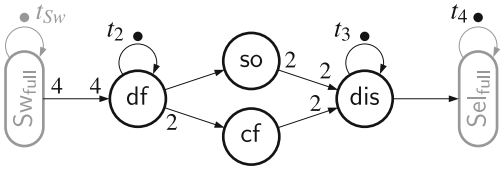


Fig. 8.19 Analysis graph, S_{full}, of the full frame scenario

executing `detect_frame_type` (dft) the next scenario has been detected and the control diverges between the scenarios. At run-time it is impossible to decide which of the scenarios is being executed until after it has started. Thus, a causality dilemma is encountered if we want to use the scenario model as an implementation model or a programming paradigm.

Because scenario graphs capture different behaviors of the same application, we argue that a given number of actors and tokens at the start of each scenario are common to all scenarios. In the example, these are actors bf and dft and tokens t_1 and t_4. After executing this maximal prefix graph the current scenario is assumed to be known. This prefix graph must be marked as such by the programmer after which it is automatically split off in a *detector scenario* det, see Fig. 8.18 (the additional actors, Sel_{det} and Sw_{det}, and additional tokens will be explained later). After execution of S_{det} the next scenario is known and can be executed, e.g., S_{full} depicted in Fig. 8.19. This solves the causality dilemma and is explained in detail in Sect. 4.2.

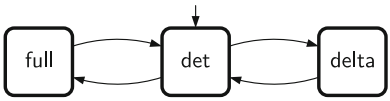


Fig. 8.20 The extended FSM with detector scenario det

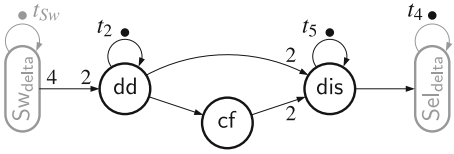


Fig. 8.21 Analysis graph, S_{delta}, of the delta frame scenario

4.2 Sequence Analysis

In Sect. 4.1 we introduced a solution to the scenario detection causality dilemma by splitting off the detector scenario det from the original scenarios, see Fig. 8.18. We assume the identification of the detector subgraph is done by the programmer. All the following steps are automated. First the FSM is transformed to execute scenario det before each transformed original scenario, see Fig. 8.20.

To model the transport of tokens from and to the detector scenario, we use *switch* and *select* actors borrowed from Boolean dataflow [5]. These are instantiated on the outgoing channels (switch) and incoming channels (select) at which the scenario graphs are split, and assigned a worst-case execution time (WCET) of zero. See Figs. 8.19 and 8.21. The switch and select actors receive Boolean control tokens from the dft frame detector. Each Sw and Sel receives a self-edge with a synchronization token that has the same label in every scenario. Token t_4 is a special case. It is available for the next scenario only after the dis actor has finished its firing and it is consumed by the bf actor of the following det scenario. Therefore t_4 is moved onto the self-edge of Sel actors of the scenario graphs, removing the need for an additional synchronization token. Additionally, we need one initial control token t_{ctrl} on the edge to the initial select actor of the detection scenario to make the graph deadlock free. The analysis graphs thus generated are shown in Figs. 8.18, 8.19, and 8.21. The Sw and Sel actors as well as newly inserted channels and tokens actors are indicated in gray.

Temporal analysis of this video decoder with SDF^3 [39] visits the scenarios as indicated by the FSM, starting with scenario det. It can be shown that the throughput of the modified graph is identical to the throughput of the original graph [48]. In each new scenario graph it is known from the start of the execution what the current scenario is, which allows the scenario graph to be directly implemented.

4.3 Scenario Execution

In this section we present an implementation for executing a sequence of scenarios that solves the following practical aspects: (*i*) *switch* and *select* implementation; (*ii*) extending static-order actor schedules on-the-fly.

4.3.1 Switch and Select Implementation

During analysis the *switch* and *select* actors ensure synchronization but function as regular SDF actors. While scenarios are analyzed separately, our implementation glues $\mathsf{S}_{\mathsf{det}}$ to the graphs of the other scenarios. In practice the switch acts as multiplexer and the select as de-multiplexer. For execution there is *one* detector subgraph serving both $\mathsf{S}_{\mathsf{full}}$ and $\mathsf{S}_{\mathsf{delta}}$ with tokens, see Fig. 8.22. Actors Sw and Sel are indicated in gray. Synchronization token t_{Sw} is not relevant for execution.

We propose a solution that exploits the `libFIFO` [19] library that implements FIFO buffers that can be disconnected and reconnected without invalidating data. A *switch* actor will have only a single output port, to which the proper channel is connected depending on the detected scenario (see Fig. 8.23). This swapping of FIFO channels is indicated with the symbol for an electrical switch, which connects the single output port either to the channel to actor df or the channel to dd. The other channel is left unconnected on one end, effectively giving it rate zero. The FIFO swap must take place before Sw fires. *Select* actors are similar but demultiplex two channels to one.

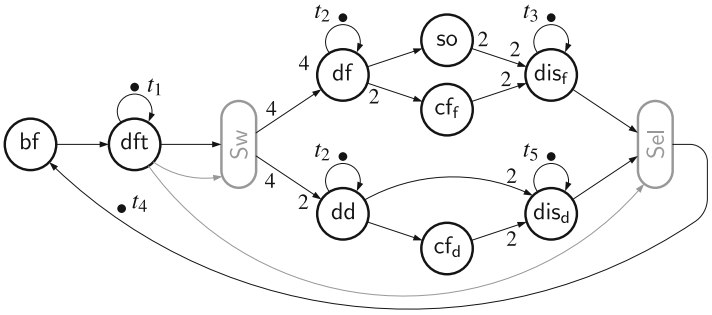


Fig. 8.22 The merged execution graph, the *switch* and *select* actors are indicated in gray

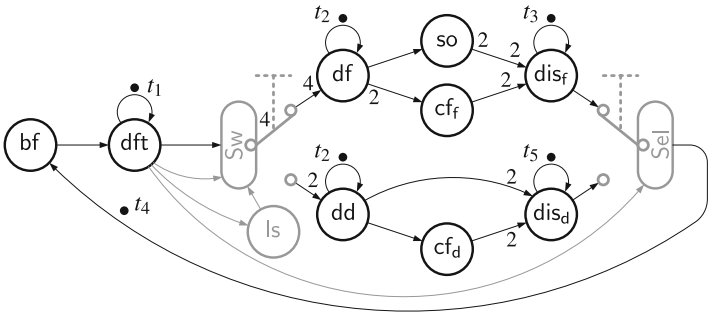


Fig. 8.23 The merged execution graph with implementation details, the *switch/select* and *load schedule* (ls) actors for a one-processor mapping are indicated in gray. The swapping of FIFO channels is indicated with the electrical symbol for a switch, also in gray. Note that this is not a valid dataflow graph

4.3.2 Extending Static-Order Schedules

We schedule actors on a processor with the `libDataflow` library [19] that executes dataflow graphs by iterating over a given static-order (SO) schedule. Such a schedule can be produced with SDF^3 [39]. The schedule blocks if an actor is not ready to fire. However, the SO schedule of our proposed solution changes depending on the detected scenario. Therefore we use a new scheduling concept that we dub the *rolling static-order* (RSO) scheduler [48].

Execution starts with $\mathsf{S}_{\mathsf{det}}$, so if we were to map the decoder to a single processor the SO schedule starts with [bf, dft, Sw]. After firing dft the next scenario is known and the SO schedule can be extended. If scenario full is detected, then the sequence [df, cf_{f}, cf_{f}, so, $\mathsf{dis}_{\mathsf{f}}$, Sel, bf, dft, Sw] is concatenated to the "rolling" schedule. Note that Sel is the last actor in the schedule of $\mathsf{S}_{\mathsf{full}}$, and we immediately concatenate the next detector scenario. This ensures that the scheduler will never run out of actors to schedule. A multi-processor mapping works similarly, the only constraint we impose is that a *switch* must be mapped onto the same processor as the actor preceding it.

The RSO scheduler has been implemented in `libDataflow` for the CompSOC platform. We initialize it with a unique SO schedule for each scenario; the start is set to $\mathsf{S}_{\mathsf{det}}$. An additional *load schedule* (ls) actor is inserted right after dft *on every processor*. See Fig. 8.23 for the single-processor example. These ls actors receive a scenario token from dft and extend the schedule accordingly. The example schedule of $\mathsf{S}_{\mathsf{det}}$ changes to [bf, dft, ls, Sw]. We furthermore exploit the ls actor to connect all FIFOs correctly before the actors Sw or Sel fire. This dependency is visualized with a gray channel in Fig. 8.23. Details on the implementation and an experimental evaluation can be found in [48].

5 Run-Time Methods

This section briefly discusses some methods that can be applied for resource management using the SADF model at run-time.

5.1 Run-Time Management

An execution platform may support for execution parameters to be changed or selected at run-time, depending on run-time state, like resource availability or variations in deadlines or slack. Such settings can be considered as run-time situations that can be characterized and represented by scenarios and correspondingly modeled with dataflow models.

The selection of different settings may lead to run-time opportunities to optimize multiple aspects of the running system. Run-time situations can be seen as partially

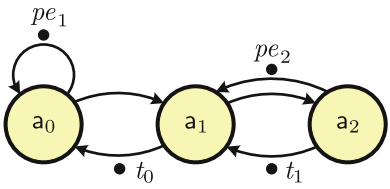


Fig. 8.24 Dataflow graph of the application scenario

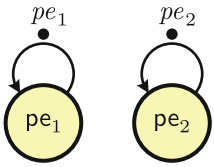


Fig. 8.25 Dataflow graph of the DVFS reconfiguration scenario

determined by the environment, e.g., by load induced by different kinds of application input, and partially by the system's run-time management. Reference [49] shows how the interplay between environment and run-time management can be seen as a two-player game, where one player is the run-time management and the opponent player is the environment. The moves in the game can be modeled as dataflow scenarios. In this case known results from the theory of two-player games, in particular, mean-payoff games and ratio games, can be exploited to determine optimal positional strategies. This leads to an optimal run-time strategy under environmental variations.

We consider a simple example taken from [9]. It considers an application shown in Fig. 8.24. The dataflow graph of the application consists of three actors, a_0, a_1, and a_2 with data dependencies captured by the tokens t_0 and t_1. It is additionally assumed that there are two processing elements, pe_1 and pe_2 and that pe_1 executes actor a_0 and pe_2 executes actors a_1 and a_2. The mapping of tasks to processing elements is modeled with the tokens pe_1 and pe_2 that capture the availability time of the processing resources pe_1 and pe_2, respectively. The processing elements support DVFS [6] to exploit a trade-off between processing speed and power consumption. We assume that there are two settings to consider, having a specific DVFS setting for each of the processing elements. We represent these two settings by two scenarios, s_1 and s_2. Both scenarios follow the application graph of Fig. 8.24, but with different actor execution times. Run-time switching between s_1 and s_2 is possible, but changing the DVFS setting for a processing element takes time and energy. The switching process is therefore modeled by a dataflow scenario itself. The graph is shown in Fig. 8.25. Note that the graph expresses that a DVFS change can only start after the processing element is available (has finished the workload of previous scenarios) and then takes some amount of time (the execution time of the actor), after which the processing element is available for processing of new workload, expressed by the production time of the token. Note that the scenario defines that both processing elements need to reconfigure, but there are no dependencies between them. They do not need to happen at the same moment, but can follow the pipelined execution of the application. We assume the reconfigurations may

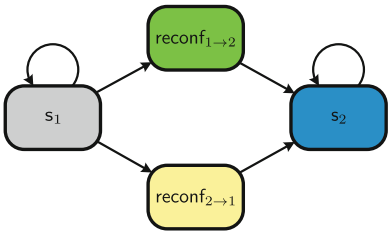


Fig. 8.26 Automaton defining the reconfiguration scenario sequences

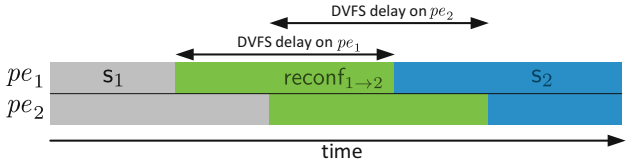


Fig. 8.27 Gantt chart of the reconfiguration

take different amounts of time and therefore there are two switching scenarios reconf$_{1\to2}$ and reconf$_{2\to1}$. The possible sequences of scenarios are defined by the automaton in Fig. 8.26. It specifies that a change of DVFS mode needs to invoke the corresponding reconfiguration scenario.

Figure 8.27 shows a Gantt chart of a fragment of the behavior including a DVFS switch from scenario s$_1$ to scenario s$_2$ by the intermediate reconfiguration scenario reconf$_{1\to2}$. The reconfiguration follows the pipelined execution of the application and pe_1 reconfigures before pe_2.

The SADF model defines all the possible behaviors that a run-time manager can choose to execute with this application. Those behaviors correspond to scenario sequences of the SADF model. The model predicts the worst-case performance of such a behavior. This can be used by an optimization strategy to find the optimal behavior in terms of power consumption that satisfies the performance constraints, for example, using model predictive control techniques.

Parametric analysis, as discussed in Sect. 3 can be particularly interesting for run-time situations. Actor execution times may only be available at run-time, but there may not be enough time or resources for a full dataflow analysis algorithm to be executed at run-time. In such a case the parametric analysis results can be applied to quickly obtain the analysis result in a particular run-time situation when the parameter values have become known. Furthermore, the parametric analysis can be utilized to achieve energy savings in a setting with varying throughput requirement. Imagine a situation where one can quantify the Quality of Service (QoS) via throughput. In most cases, the higher the throughput, the higher the QoS and vice versa. On the other hand, achieving a higher or lower throughput, in terms of dataflow, means that the actors will have to attain shorter or longer firing delays, respectively. The question is, given a run-time change in QoS expressed as a throughput constraint, how long is short enough to meet the throughput constraint. Reference [9] discusses a heuristic strategy to find such optimal dynamic voltage

and frequency scaling (DVFS) settings using the parametric throughput expression of the model of the application. In Chap. 9 it is shown how this strategy can also be applied to optimal controller synthesis problems for flexible manufacturing systems.

6 Conclusions

This chapter has discussed how the scenario methodology can be applied to and combined with dataflow modeling and analysis. The resulting model SADF is an expressive model that can express variation in functional behavior and consequently diverse run-time situations. Dataflow models can also be used in the context of real-time streaming applications as a formal abstraction of concrete realizations that guarantees preservation of performance for its refinements.

We have detailed the model and its semantics and have described common performance analysis methods for throughput and latency, including parametric analysis techniques. Timed dataflow is a performance oriented model in a wider class of max-plus linear systems. The properties of max-plus linear systems have been discussed to better understand the essential properties of systems to be amenable to being modeled with max-plus linear abstractions.

We have also shown how the SADF model can be applied as a programming model to ensure a good match between an implementation on a multi-processor platform and the abstract dataflow model. It was shown how to integrate the scenario detection methods and derive an accurate model of a causal implementation of the behavior. An online static-order scheduling technique is presented to realize the implementation.

We have additionally illustrated how the dataflow model and the parametric analysis can be combined to realize efficient run-time management strategies.

Acknowledgements This research is supported in part by the ARTEMIS joint undertaking through the ALMARVI project (621439) and by the ITEA3 project 14014 ASSUME.

References

1. H.A. Ara, A. Behrouzian, M. Hendriks, M. Geilen, D. Goswami, T. Basten, Scalable analysis of multi-scale dataflow models. ACM Trans. Embed. Comput. Syst. **16**(4), 80:1–80:26 (2018)
2. F. Baccelli, G. Cohen, G., G. Olsder, J.P. Quadrat, *Synchronization and Linearity* (Wiley, London, 1992)
3. S.S. Battacharyya, E.A. Lee, P.K. Murthy, *Software Synthesis from Dataflow Graphs* (Kluwer Academic Publishers, Norwell, 1996)
4. T. Bijlsma, M. Bekooij, G. Smit, Circular buffers with multiple overlapping windows for cyclic task graphs, in ed. by P. Stenström. *Transactions on High-Performance Embedded Architectures and Compilers III*. Lecture Notes in Computer Science (Springer, Berlin, 2011). https://doi.org/10.1007/978-3-642-19447-4

5. J.T. Buck, Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, EECS Department, University of California, Berkeley, 1993
6. A.P. Chandrakasan, S. Sheng, R.W. Brodersen, Low-power CMOS digital design. IEEE J. Solid State Circuits **27**(4), 473–484 (1992). https://doi.org/10.1109/4.126534
7. J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. Gettrick, J.P. Quadrat, Numerical computation of spectral elements in max-plus algebra, in *Proceedings of the IFAC Conference on System Structure and Control* (Nantes, 1998)
8. M. Damavandpeyma, S. Stuijk, M. Geilen, T. Basten, H. Corporaal, Parametric throughput analysis of scenario-aware dataflow graphs, in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pp. 219–226 (IEEE, Piscataway, 2012). https://doi.org/10.1109/ICCD.2012.6378644
9. M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, H. Corporaal, Throughput-constrained DVFs for scenario-aware dataflow graphs, in, *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (IEEE, Piscataway, 2013), pp. 175–184. https://doi.org/10.1109/RTAS.2013.6531090
10. A. Dasdan, R.K. Gupta, Faster maximum and minimum mean cycle algorithms for system-performance analysis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **17**(10), 889–899 (1998). https://doi.org/10.1109/43.728912
11. V. Dhingra, S. Gaubert, How to solve large scale deterministic games with mean payoff by policy iteration, in *Proceedings of the 1st International Conference on Performance Evaluation Methodologies and Tools, Valuetools '06* (ACM, New York, 2006). https://doi.org/10.1145/1190095.1190110
12. S. Gaubert, Performance evaluation of (max, +) automata. IEEE Trans. Autom. Control **40**(12), 2014–2025 (1995)
13. M. Geilen, If we could go back in time…on the use of 'unnatural' time and ordering in dataflow models, in ed. by M. Lohstroh, P. Derler, M. Sirjani. *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday* (Springer International Publishing, Cham, 2018), pp. 267–286. https://doi.org/10.1007/978-3-319-95246-8_16
14. M. Geilen, T. Basten, Requirements on the execution of Kahn process networks, in ed. by P. Degano. *Proceedings of the 12th European Symposium on Programming, ESOP 2003*. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7–11, 2003. Lecture Notes in Computer Science, vol. 2618 (Springer, Berlin, 2003)
15. M. Geilen, S. Stuijk, Worst-case performance analysis of synchronous dataflow scenarios, in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (ACM, New York, 2010), pp. 125–134
16. M. Geilen, S. Tripakis, M. Wiggers, The earlier the better: A theory of timed actor interfaces, in Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, HSCC '11 (ACM, New York, 2011), pp. 23–32
17. M. Geilen, J. Falk, C. Haubelt, T. Basten, B. Theelen, S. Stuijk, Performance analysis of weakly-consistent scenario-aware dataflow graphs. J. Signal Process. Syst. **87**(1), 157–175 (2017). https://doi.org/10.1007/s11265-016-1193-7
18. A.H. Ghamarian, M.C.W. Geilen, T. Basten, S. Stuijk, Parametric throughput analysis of synchronous data flow graphs, in *2008 Design, Automation and Test in Europe* (IEEE, Piscataway, 2008), pp. 116–121. https://doi.org/10.1109/DATE.2008.4484672
19. K. Goossens, A. Azevedo, K. Chandrasekar, M.D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A.B. Nejad, A. Nelson, S. Sinha, Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. SIGBED Rev. **10**(3), 23–34 (2013). https://doi.org/10.1145/2544350.2544353
20. R. Gu, J.W. Janneck, M. Raulet, S.S. Bhattacharyya, Exploiting statically schedulable regions in dataflow programs. J. Signal Process. Syst. **63**(1), 129–142 (2011). https://doi.org/10.1007/s11265-009-0445-1
21. B. Heidergott, G.J. Olsder, J. van der Woude, *Max Plus at Work* (Princeton University Press, Princeton, 2006)

22. C.A.R. Hoare, Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978) https://doi.org/10.1145/359576.359585
23. A. Jantsch, *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation* (Morgan Kaufmann Publishers, San Francisco, 2003)
24. G. Kahn, The semantics of a simple language for parallel programming, in ed. by J. Rosenfeld. *Information Processing 74: Proceedings of the IFIP Congress*, vol. 74, Stockholm, Sweden, August 1974 (North-Holland, Amsterdam, 1974), pp. 471–475
25. E. Lee, D. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **C-36**(1), 24–35 (1987). https://doi.org/10.1109/TC.1987.5009446
26. T. Lundqvist, P. Stenström, Timing anomalies in dynamically scheduled microprocessors, in *Proceedings of the 20th IEEE Real-Time Systems Symposium, RTSS '99* (IEEE Computer Society, Washington, 1999), pp. 12. http://dl.acm.org/citation.cfm?id=827271.829103
27. A. Moonen, M. Bekooij, R. van den Berg, J.L. van Meerbergen, Practical and accurate throughput analysis with the cyclo static dataflow model, in 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2007) (IEEE, Piscataway, 2007), pp. 238–245. http://dx.doi.org/10.1109/MASCOTS.2007.52
28. O. Moreira, Temporal analysis and scheduling of hard real-time radios running on a multiprocessor. Ph.D. thesis, Eindhoven University of Technology, 2012
29. O. Moreira, H. Corporaal, *Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor* (Springer, Berlin, 2014)
30. O. Moreira, T. Basten, M. Geilen, S. Stuijk, Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. IEEE Trans. Comput. **59**(2), 188–201 (2010). https://doi.org/10.1109/TC.2009.155. Cited By 24
31. S. Neuendorffer, E. Lee, Hierarchical reconfiguration of dataflow models, in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04* (IEEE, Piscataway, 2004), pp. 179–188. https://doi.org/10.1109/MEMCOD.2004.1459852
32. T. Parks, Bounded Scheduling of Process Networks. Ph.D. thesis, University of California, EECS Dept., Berkeley, CA, 1995
33. F. Siyoum, M. Geilen, J. Eker, C. von Platen, H. Corporaal, Automated extraction of scenario sequences from disciplined dataflow networks, in *2013 Eleventh IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)* (IEEE, Piscataway, 2013), pp. 47–56
34. M. Skelin, Worst-case performance analysis of scenario-aware real-time streaming applications. Ph.D. thesis, Norwegian University of Science and Technology (NTNU), 2016
35. M. Skelin, M. Geilen, Compositionality in scenario-aware dataflow: a rendezvous perspective, in *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Association for Computing Machinery, New York, 2018), pp. 55–64
36. M. Skelin, M. Geilen, F. Catthoor, S. Hendseth, Parameterized dataflow scenarios. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **36**(4), 669–682 (2017). https://doi.org/10.1109/TCAD.2016.2597223
37. M. Skelin, M. Geilen, F. Catthoor, S. Hendseth, Worst-case performance analysis of sdf-based parameterized dataflow. Microprocess. Microsyst. **52**, 439–460 (2017). https://doi.org/10.1016/j.micpro.2016.12.004
38. S. Sriram, S.S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd edn. (CRC Press, Boca Raton, 2009)
39. S. Stuijk, M. Geilen, T. Basten, SDF^3: SDF for free, in *Proceedings of the 6th International Conference on Application of Concurrency to System Design, ACSD 2006* (IEEE Computer Society Press, Los Alamitos, 2006), pp. 276–278. http://www.es.ele.tue.nl/sdf3

40. S. Stuijk, M. Geilen, T. Basten, Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Trans. Comput. **57**(10), 1331–1345 (2008). http://dx.doi.org/10.1109/TC.2008.58
41. S. Stuijk, M. Geilen, T. Basten, A predictable multiprocessor design flow for streaming applications with dynamic behaviour, in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools* (IEEE, Piscataway, 2010), pp. 548–555. https://doi.org/10.1109/DSD.2010.31
42. S. Stuijk, M. Geilen, B. Theelen, T. Basten, Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications, in *2011 International Conference on Embedded Computer Systems (SAMOS)* (IEEE, Piscataway, 2011), pp. 404–411. https://doi.org/10.1109/SAMOS.2011.6045491
43. B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, S. Stuijk, A scenario-aware data flow model for combined long-run average and worst-case performance analysis, In: *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06* (IEEE, Piscataway, 2006), pp. 185–194. https://doi.org/10.1109/MEMCOD.2006.1695924
44. T. van den Boom, B. De Schutter, Modelling and control of discrete event systems using switching max-plus-linear systems. Control. Eng. Pract. **14**(10), 1199–1211 (2006). https://doi.org/10.1016/j.conengprac.2006.02.006
45. B. van der Sanden, Performance analysis and optimization of supervisory controllers. Ph.D. thesis, Eindhoven University of Technology, 2018
46. B. van der Sanden, J. Bastos, J. Voeten, M. Geilen, M.A. Reniers, T. Basten, J. Jacobs, R.R.H. Schiffelers, Compositional specification of functionality and timing of manufacturing systems, in *2016 Forum on Specification and Design Languages, FDL 2016*, (IEEE, Piscataway, 2016), pp. 1–8. https://doi.org/10.1109/FDL.2016.7880372
47. R. van Kampenhout, S. Stuijk, K. Goossens, A scenario-aware dataflow programming model, in *2015 Euromicro Conference on Digital System Design (DSD)* (IEEE, Piscataway, 2015), pp. 25–32. https://doi.org/10.1109/DSD.2015.28
48. R. van Kampenhout, S. Stuijk, K. Goossens, Programming and analysing scenario-aware dataflow on a multi-processor platform, in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)* (IEEE, Piscataway, 2017)
49. Y. Yang, M. Geilen, T. Basten, S. Stuijk, H. Corporaal, Playing games with scenario- and resource-aware SDF graphs through policy iteration, in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)* (2012), pp. 194–199. https://doi.org/10.1109/DATE.2012.6176462

Chapter 9
Scenarios in the Design of Flexible Manufacturing Systems

Twan Basten, João Bastos, Róbinson Medina, Bram van der Sanden, Marc C. W. Geilen, Dip Goswami, Michel A. Reniers, Sander Stuijk, and Jeroen P. M. Voeten

1 Introduction

Today's flexible manufacturing systems (FMS) are rapidly developing into complex cyber-physical systems (CPS) that show a tight coupling between cyber aspects (embedded software and electronics) and physical aspects (related to mechanics, such as acceleration and deceleration of motors, and physical processes, such as vibrations, heating, and cooling). An example of such a CPS is a professional printer like the Océ VarioPrint i300 [28] that prints 150 duplex A4 color pages per minute while guaranteeing that the paper is not damaged in the transport and guaranteeing print quality by appropriate heating and cooling of the sheets of paper. Other examples are lithography machines that expose patterns of electronic circuits on wafers with very high positional accuracy, interventional X-ray machines for minimally invasive surgery guided by X-ray images, or electron microscopes that produce extremely detailed images with a granularity of individual atoms. Such FMS should continuously operate to maximize production in terms of completed products (sheets, wafers, images) per time unit. Productivity is limited by the

T. Basten (✉)
Eindhoven University of Technology and ESI, TNO, Eindhoven, The Netherlands
e-mail: a.a.basten@tue.nl

J. Bastos · R. Medina · B. van der Sanden · M. C. W. Geilen · D. Goswami · M. A. Reniers · S. Stuijk
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: j.p.nogueira.bastos@tue.nl; r.a.medina.sanchez@tue.nl; b.v.d.sanden@tue.nl; m.c.w.geilen@tue.nl; d.goswami@tue.nl; m.a.reniers@tue.nl; s.stuijk@tue.nl

J. P. M. Voeten
Eindhoven University of Technology, Eindhoven, The Netherlands

ESI, TNO, Eindhoven, The Netherlands
e-mail: j.p.m.voeten@tue.nl

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6_9

physical design of the system (determining, for example, the time needed to move (partial) products between locations), the control software operating the system (determining, for example, the speed of heating objects in heaters or the speed of moving robot arms manipulating objects), and the processing of large amounts of data (for example, in image-based control loops). But FMS typically have stringent productivity requirements, directly related to profitability. Optimizing productivity during early design phases is important for system cost and engineering efficiency. Therefore, it is key to have effective techniques to model, analyze, and optimize system productivity.

We advocate that model-based design in combination with a V-model-based performance engineering method [18] is suitable to address performance challenges in early design phases. There are many interesting questions related to model-based performance engineering of FMS. Some examples: Can we accurately predict FMS productivity early in the design process? How do we identify productivity bottlenecks and explore design alternatives? Can we develop supervisory control software that not only guarantees correct behavior but also optimizes productivity? And can we co-optimize the data-intensive control loops in terms of quality of control and implementation efficiency? In this chapter, we present scenario-based techniques addressing these challenges.

Section 2 introduces a motivating case study. Section 3 discusses related work on FMS modeling and analysis. Section 4 introduces scenario-based models for FMS. The operation of FMS can be broken down into deterministic activities (like moving an object from one position to another one) that exhibit little or no timing variation. Each activity may consist of multiple actions (like moving a robot arm in a specific direction for a certain distance). Activity executions correspond to run-time situations as introduced in Chap. 2. Activities thus form system scenarios. Activity models can be naturally analyzed for performance using max-plus algebra [5, 13], along the same lines as the dataflow analysis introduced in Chap. 8. This analysis setup provides a basis for early design-space exploration. Section 5 discusses supervisory controller synthesis and optimization based on activity models. Allowed activity orderings can be specified by (extended) finite-state machines (FSM). A safe, deadlock-free and maximally permissive supervisory controller can be synthesized from a collection of such models, capturing different requirements on permitted activity orderings. Such a supervisory controller can then be optimized for performance by considering the timing information of scenarios. The performance analysis and controller synthesis of Sects. 4 and 5 share a common scenario-based modeling approach. Section 6 presents another, independent use of scenarios in FMS optimization, namely scenario-based optimization of data-intensive control loops through adaptive pipelined sensing. Data-intensive control loops may have a long sensing latency. Parallelized pipelined sensing exploiting today's multi-core technology allows to increase the sampling rate of the controller despite this long data processing latency, improving the achievable quality of control (QoC). Each pipelining configuration corresponds to a system scenario in the framework of Chap. 2. Interestingly, the impact of the improved QoC of a control loop in the FMS on the system productivity can in turn be analyzed with appropriate activity models, using the techniques of Sects. 4 and 5. Section 7 concludes the chapter.

2 Motivating Case Study: xCPS

xCPS (eXplore CPS), see Fig. 9.1, is an assembly line emulator that can be used for research and education in the CPS domain [1]. The machine processes two types of cylindrical pieces, tops and bottoms, of different colors, see Fig. 9.2; the main use case of xCPS is to compose tops and bottoms into composite products, like the ones shown in the top left of Fig. 9.2.

Figure 9.3 shows a schematic view of the layout of xCPS. Pieces enter the machine at the top right, on conveyer belt 1; composite products leave the machine at the bottom right, via conveyer belt 5. xCPS consists of one storage area, six conveyor belts, two indexing tables, two gantry arms, and several actuators and sensors. The storage area is a grid where 25 objects (top or bottom pieces, or composites) can be stored. Six stopper actuators in strategic positions along the conveyer belts can stop the movement of objects, effectively creating buffers accumulating objects on the conveyor belts. Switches enable route changes for individual objects. xCPS has

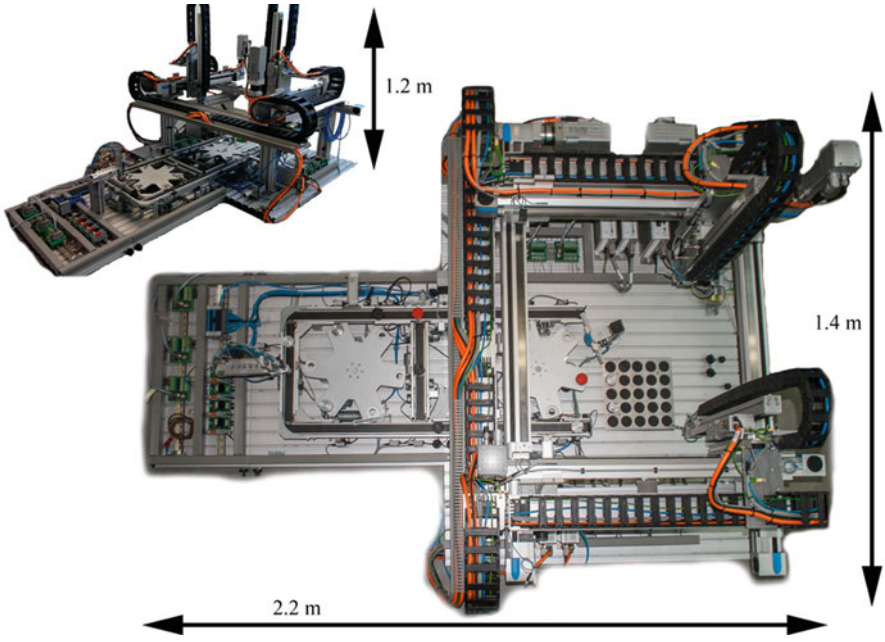


Fig. 9.1 xCPS—an assembly line emulator

Fig. 9.2 Assembly pieces in xCPS

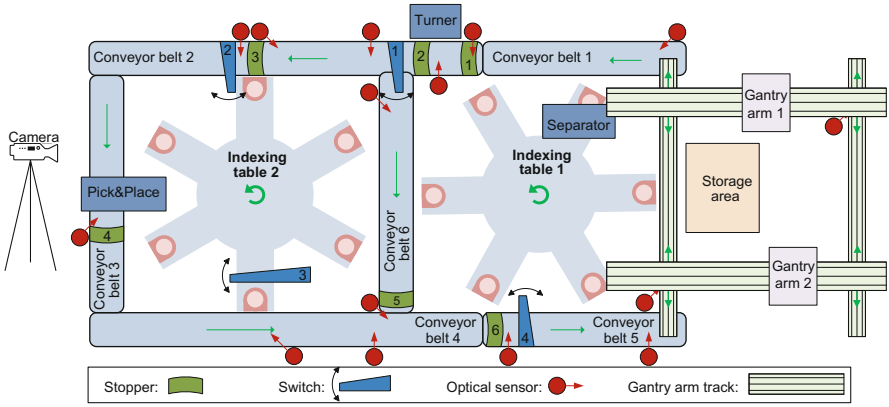


Fig. 9.3 xCPS system layout

a turner to flip pieces. Moreover, it has two actuators for assembly and disassembly of pieces. The pick and place actuator can clamp a piece and combine it with a complementary piece. The separator can disassemble a composite into its two constituent pieces. xCPS has 15 sensors that can detect the presence of an object. These sensors cannot distinguish the type of an object, its color, or its exact location. To detect the type, color, and location of an object, a camera can be added to the setup.

There are many possible use cases of the xCPS platform. Besides the already mentioned main use case and other production use cases, the gantry arms and storage area can be used as a sorting station, where the storage is sorted according to selected criteria, but it can also be used for games such as tic-tac-toe and four-in-a-row. Each use case comes with its own design challenges.

xCPS can be used to test and evaluate CPS performance engineering solutions for such use cases, addressing questions like the ones already posed in the introduction. Suitable models are needed to predict productivity of the xCPS assembly process producing composites from bottom and top pieces and to detect performance bottlenecks. Supervisory controllers should ensure safety of operation, avoiding, for instance, gantry arm collisions, and avoid deadlocks, due to, for example, full buffers. Productivity should be optimized by proper routing of the flows of pieces, proper scheduling of the actions, and appropriate settings of the belt speeds and control of the various actuators in the system. Image-based control may be used to optimize settings online during operation.

3 Related Work

Models of FMS, and CPS in general, may serve many different purposes and take many different shapes. A recent book of Alur [2] provides a comprehensive overview of specification, modeling, and analysis techniques that play a role in CPS design.

When focusing on performance engineering of CPS, traditionally, performance questions are addressed using discrete-event simulation or hybrid discrete/continuous-time simulation, see, e.g., [6, 19, 20, 22, 27, 31, 34, 45]. Discrete-event simulations capture all relevant events in the system. Hybrid discrete/continuous-time simulations moreover capture the dynamics of relevant physical processes in differential equations. For complex systems, such simulations are time consuming. They are useful when investigating specific behaviors and timing aspects, like average throughput or buffer occupancy under input variations, or the performance of a specific controller; they are less suitable for exploring large design spaces. The Petri-net formalism was one of the first formalisms proposed for discrete-event simulation and analysis of FMS [27, 34, 45]. The combination of a graphical representation with a sound mathematical basis made Petri nets very popular for a wide range of domains. Chi [6] is a process-algebraic language for specifying and analyzing hybrid systems. It has a rigorous mathematical semantics and focuses on modular specification and analysis. In [19, 20], a refinement-based design method based on the POOSL language and discrete-event simulator is proposed. The method covers multiple model abstractions, ranging from high-level event models without timing information to detailed models including timing. These detailed timing models serve as a basis for synthesis of real-time software that is guaranteed to adhere to the specified timing properties. Simulink [25] is the industrial state-of-practice for discrete-event and hybrid simulation. Reference [31] describes a Simulink/SimEvents modeling approach for evolvable production systems, based on discrete-event agent models of system components like robot arms and conveyer belts. A case study illustrates the analysis of the assembly time for a simple assembly system. Potential integration with continuous-time models and code generation are mentioned as strengths of the approach; high modeling effort and poor scalability of required simulation time are mentioned as limitations.

When the goal of performance analysis is to explore the design space of possible FMS configurations towards productivity optimization, simulation is often less suitable. It is practically intractable to cover many design alternatives through detailed simulations. Moreover, given non-deterministic elements in an FMS, such as varying input or errors in sensing or actuation, it is hard to provide full coverage over all possible behaviors of any given design alternative. This implies that it is difficult, if not impossible, to decide on the best possible FMS design configuration and to provide guarantees about the achieved productivity.

Model checking [30] is a class of techniques that essentially exhaustively explores a model to prove or refute a property of interest. So model checking can be used to provide performance guarantees for FMS. In [17], for example, model

checking using the Uppaal tool is used to design controllers for a wafer scanner that avoid deadlocks and optimize throughput (i.e., productivity). Timed automata [3] and its derivatives like priced timed automata [9], supported by tools like Uppaal [38], provide state-of-the-art model checking techniques for timed systems. The already mentioned Simulink [25] and Chi [6] tool sets also support model checking and other types of formal verification techniques providing guarantees. However, model checking techniques do not scale well to complex systems with many possible behaviors and large state spaces [44]. They are therefore also not very well suited for exploring the design space of such systems.

For design-space exploration for performance optimization, compact, high-level performance models that provide conservative, tight bounds on system productivity are needed. Activity [42] and dataflow [11] models building on the foundations of (max,+) algebra [5, 13] and scenario-aware dataflow (SADF) [7, 36] have been developed for this purpose. In the remainder, we show how activity models can be used for performance analysis and design-space exploration, and for the synthesis of throughput-optimal supervisory controllers. Section 6 uses these models to illustrate the impact of improved motor control on system-level productivity. The timing information in the system-level activity models used for comparison of different controllers is extracted from detailed Simulink simulations of the controllers.

4 Activity Modeling and Performance Analysis of FMS

System-level FMS behavior can be described compactly in *activities* [42] capturing end-to-end deterministic system operations, such as the processing of a piece in the xCPS case. An activity consists of atomic system *actions* such as the activation/deactivation of an actuator and dependencies among those actions. Multiple activities together capture more elaborate functional behaviors such as the complete manufacturing of a product. Orderings among actions are enforced through resource claims and releases. This allows concurrent execution of actions of different activities that do not share resources. The different possible activity orderings within the system can be captured in a finite-state machine (FSM).

The timing semantics of an activity is concisely described by a (max,+) matrix. Such a matrix captures the (worst-case) timing behavior induced by action execution inside an activity and the dependencies among those actions. A (max,+) automaton [13] can then be derived to describe the timing semantics of the complete system. This formalization enables system-level throughput analysis to find a lower-bound guarantee on the system productivity within the set of possible activity orderings of the system.

Activities capture behavioral fragments of an FMS with an end-to-end deterministic (worst-case) timing. Activities thus correspond to system scenarios. An activity-based specification can be converted to the FSM-SADF models [7] of the previous chapter, by mapping each activity onto an SADF scenario. The underlying (max,+) timing semantics and analyses are common to both modeling approaches.

The above motivates a scenario-based design method in which designers of FMS specify system-level behavior as activities and their possible orderings. This allows early, model-based design-space exploration of the FMS design to study the performance impact of different choices as well as identifying possible bottlenecks in the manufacturing flow. This section describes the concepts of the modeling approach, giving illustrative examples from the xCPS case study introduced in Sect. 2.

4.1 Activities

An FMS is decomposed into a set of *peripherals* (P), a set of *actions* (A), and a set of *resources* (R). A peripheral can execute actions. An action describes an atomic behavior of the system, e.g., the movement of a motor or the actuation of an on/off peripheral such as a gripper. The complete set of actions describes all behavior that the system can exhibit. Peripherals are aggregated into *resources*, which can be *claimed* and *released*. As an example, consider the Turner resource of the xCPS system shown in Fig. 9.1, which can flip the orientation of top and bottom pieces. This resource is composed of three peripherals: a gripper (which holds the piece), a motor (which is able to lift and flip the piece), and a position sensor (which provides information on the position of the turner). Actions of these peripherals can be a single motor translation, the gripping of a piece, or the indication that the turner is up.

Using peripheral actions, deterministic functional behaviors of the system can be constructed as *activities*. An activity is a directed acyclic graph (DAG) consisting of a set N of nodes and a set $\rightarrow \subseteq N \times N$ of (ordering) dependencies between nodes. Nodes refer to either an action executed by a peripheral (associated with a pair (a, p) with $a \in A$ and $p \in P$) or a claim (cl) or release (rl) of a resource (associated with a pair (r, v) with $r \in R$ and $v \in \{cl, rl\}$). Activities have to satisfy a number of static consistency constraints. These ensure that proper claiming and releasing of resources is respected within an activity.

- All nodes referring to the same peripheral are sequentially ordered by the dependencies (to avoid self-concurrency which is physically not possible).
- Each resource is claimed no more than once; each resource is released no more than once.
- Every action node is preceded by a claim on the corresponding resource and succeeded by a release of the corresponding resource.
- Every release is preceded by a claim on the corresponding resource; every claim is succeeded by a release of the corresponding resource.

xCPS Case Study 1 Figure 9.4 depicts the activity *Bottom* which captures the processing of a bottom piece in xCPS. It has several parameters that are clarified in the following. Actions are represented as nodes and dependencies as edges, where

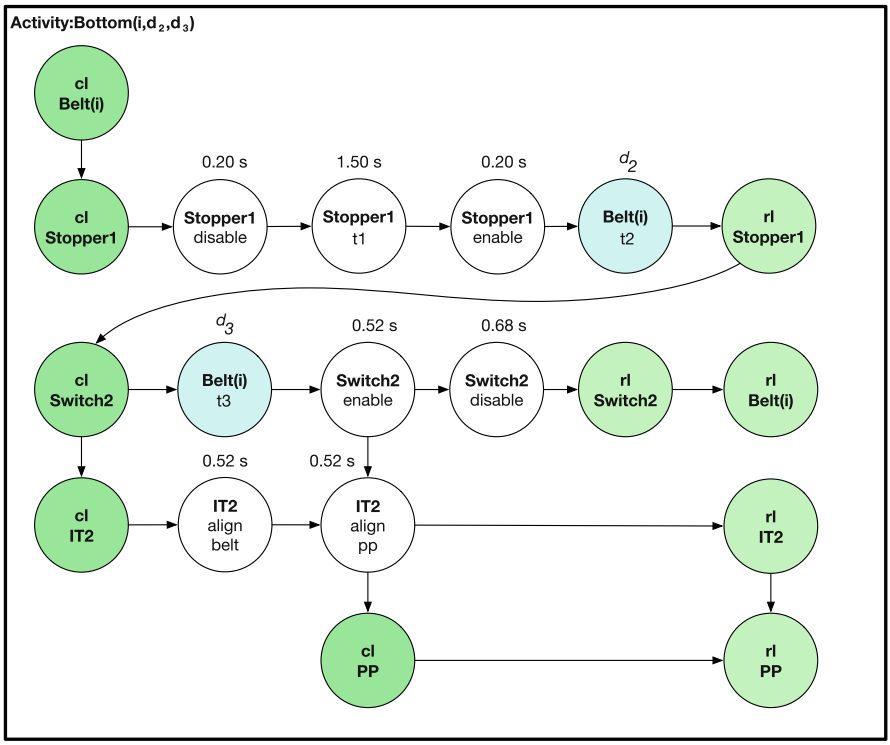


Fig. 9.4 Activity *Bottom*(i, d_2, d_3) capturing the processing of a bottom piece

green colored nodes refer to resource claims and releases. White and blue nodes are action nodes, labeled with resource and action names inside the node. Resource names are used instead of peripheral names, to ease readability. Action nodes are further annotated with their (worst-case) execution times. Timing in activity models is explained in detail in the next section. Blue actions have execution times that may vary, depending on the pieces entering the system before and after the modeled bottom piece. The variants are captured through parameters d_2 and d_3.

The goal of our models is to predict system-level productivity of xCPS. That is, we want to know how many composite products are produced per second. The activity model of Fig. 9.4 captures the input of a bottom piece and its route through the system. For our analyses, we make the following assumptions[1]:

[1]The assumptions on the one hand simplify the actual xCPS case to allow its use as a running example throughout the chapter; on the other hand, some assumptions (like the belt speeds and certain sensor and motor functionality) deviate from the actual system so that it is possible to illustrate all concepts introduced in this chapter.

- All switches and stoppers of xCPS are initially disabled, except for Stopper 1 which is initially enabled. (Figure 9.3 shows switches and stoppers in enabled state.)
- Pieces enter the system at Stopper 1;
- Belt 1 is providing pieces sufficiently fast and does not play a role in system-level performance.
- An alternating sequence of bottom and top pieces flows through the system, starting with a bottom. A new piece can enter the system only once the previous piece passed the Turner unit. Bottoms are always properly oriented. Tops may enter the system in inverted orientation; these are then flipped by the Turner.
- Bottoms move to the pick and place (PP) unit through indexing table 2 (IT2); tops move to the pick and place unit over the belts.
- Rotation speed of IT2 and belt speeds are such that transportation of bottoms via IT2 is much faster than transportation of tops via the belts. This allows us to simplify the IT2 behavior, restricting it to 90° rotation actions.
- After assembly of a top and bottom, the composite piece leaves the system via IT2 and Belts 4 and 5; the timing of this part does not play a role in system-level performance.
- A maximum of three pieces are allowed on the combination of Belts 2 and 3 simultaneously.
- Belts run at a constant speed, except for Belt 2, which needs to run at a (constant) lower speed when tops are being inverted or bottoms are moved onto the indexing table. (In this setup, Stoppers 2 and 3 are not needed; a controller adapts the belt speeds based on sensor signals.)
- Actions t1–t4 correspond to transporting pieces over Belts 2 and 3 through Stopper 1, through the Turner section, towards Switch 2, and towards the pick and place unit, respectively.

Considering again the bottom activity model in Fig. 9.4, we see that a bottom piece first flows through Stopper 1 over Belt 2 to Switch 2. Stopper 1 is only disabled to allow a next piece on Belt 2 after the bottom piece has been transported through the Turner section, with the t2 action. In the actual system, actions that depend on the arrival of pieces to certain belt locations are triggered by related sensor signals. We model system dynamics such as sensor triggers, or other such events, with time abstractions such as the t2 action. After arriving at Switch 2, via the t3 transport action, the bottom piece is pushed onto indexing table IT2. IT2 is aligned with the belt before the piece is pushed onto it; afterwards, it is aligned with the pick and place (PP) unit where a top piece is placed on top of the bottom piece to assemble a composite piece. The two align actions correspond to 90° rotations. Assembly is a separate activity introduced below. After properly aligning IT2, the PP unit is claimed and released to model the arrival of the bottom piece at the PP unit. The belt capacity of three pieces is modeled through three resources Belt(i), with $i \in \{1, 2, 3\}$. A bottom activity needs any one of these three resources for its transport over Belt 2. In Sect. 4.3, where the modeling and analysis of activity sequences is

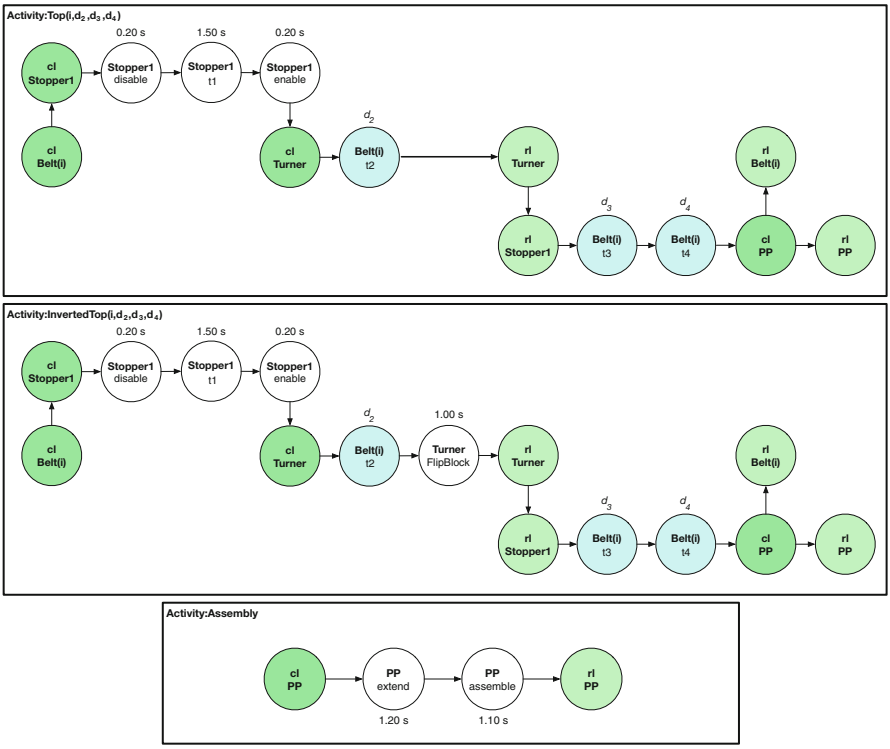


Fig. 9.5 Activities *Top*(i, d_2, d_3, d_4), *InvertedTop*(i, d_2, d_3, d_4), and *Assembly*

discussed, it becomes clear that subsequent input pieces claim these three resources in order in a cyclic manner.

Figure 9.5 shows *Top*, *InvertedTop*, and *Assembly* activities. Top pieces flow through the machine over the belts until the pick and place unit PP. The only difference between a Top and an InvertedTop activity is that an inverted piece is flipped by the Turner. The arrival of a top piece at the PP unit is again signaled by claiming and releasing the PP resource. The actual assembly is modeled by the *Assembly* activity. The PP unit picks up a top and extends towards the indexing table, before combining the top with the bottom piece on the indexing table. ■

As illustrated by the xCPS example, an activity represents a block of deterministic functional behavior. It is a design choice how much behavior should be included in a single activity. As an extreme case, an activity could be defined as a single action (together with the corresponding associated resource claim and release). On the other hand, a single activity could also be defined such that it encompasses all the behavior in the system (as long as the behavior is functionally deterministic). The most suitable activity design is system dependent, for instance, based on the dynamics of the system or a desired system architecture.

4.2 Temporal Semantics

Besides the functional specification of activities of an FMS, for productivity analysis and optimization, we need a temporal specification in the form of action execution times. A function $T_a : A \to \mathbb{R}_{\geq 0}$ maps each action to its non-negative execution time. In the activity graphs of Figs. 9.4 and 9.5, action execution times are shown above the action nodes. In reality, action execution times often show (small) variations. These variations are abstracted to a fixed execution time, typically the worst-case execution time when the purpose of the modeling is to analyze productivity guarantees. In system scenario terminology, run-time situations with small timing variations are abstracted into scenarios.

Since actions are executed on resources, actions from different activities may compete for those resources. Performance of an FMS therefore is determined by the combination of action execution times and claims and releases of resources. An action can execute if the activity of which it is a part has a claim on the resource needed by the action and if all actions preceding it have executed. For the purpose of performance analysis, system state can therefore be captured by resource-availability times; progress is captured by the aggregated duration of action executions in an activity.

xCPS Case Study 2 Consider again Fig. 9.4 depicting the bottom activity. Consider for the purpose of this example activity *Bottom*(1, 4.24, 4.57), i.e., the instance with $i = 1$, $d_2 = 4.24$, and $d_3 = 4.57$. Assume that resources Stopper1, Switch2, IT2, PP, Turner, Belt(1), Belt(2), and Belt(3) are all available at time 0. This is compactly represented by vector $[0, 0, 0, 0, 0, 0, 0, 0]^T$, or $\mathbf{0}$ for short. (The other resources in xCPS, like Switch 1, do not play a role in our performance analysis and are abstracted away.) Assume that claims and releases of resources do not take time.

From Fig. 9.4, it can be seen that resource Stopper1 is released $0.20 + 1.50 + 0.20 + 4.24 = 6.14$ time units after it has been claimed. Moreover, the release of Switch2 takes place $4.57 + 0.52 + 0.68 = 5.77$ time units after it has been claimed. Given the assumption that all resources are available at time 0, Stopper1 is released at time 6.14. Since Switch2 can only be claimed after the release of Stopper1, Switch2 is effectively claimed at time 6.14 and released at time 11.91.

Based on similar reasoning, Belt(1) is released at time 11.91 and IT2 is claimed at time 6.14 and released at time $6.14 + 4.57 + 0.52 + 0.52 = 11.75$. The latter is obtained by taking the time Switch2 is claimed and adding the maximum duration among the action sequences leading from this claim to the release of IT2. PP is claimed and released at time 11.75.

Summarizing these analyses yields the resource-availability times after execution of activity *Bottom*(1, 4.24, 4.57): $[6.14, 11.91, 11.75, 11.75, 0, 11.91, 0, 0]^T$ in vector form. The three resources that are not used by the bottom activity keep their availability time of 0. Thus, executing activity *Bottom*(1, 4.24, 4.57) turns initial state $\mathbf{0}$ into state $[6.14, 11.91, 11.75, 11.75, 0, 11.91, 0, 0]^T$. Note that the assumptions on activity models guarantee that all resources are properly released after executing an activity, so all resources have a well-defined availability time after

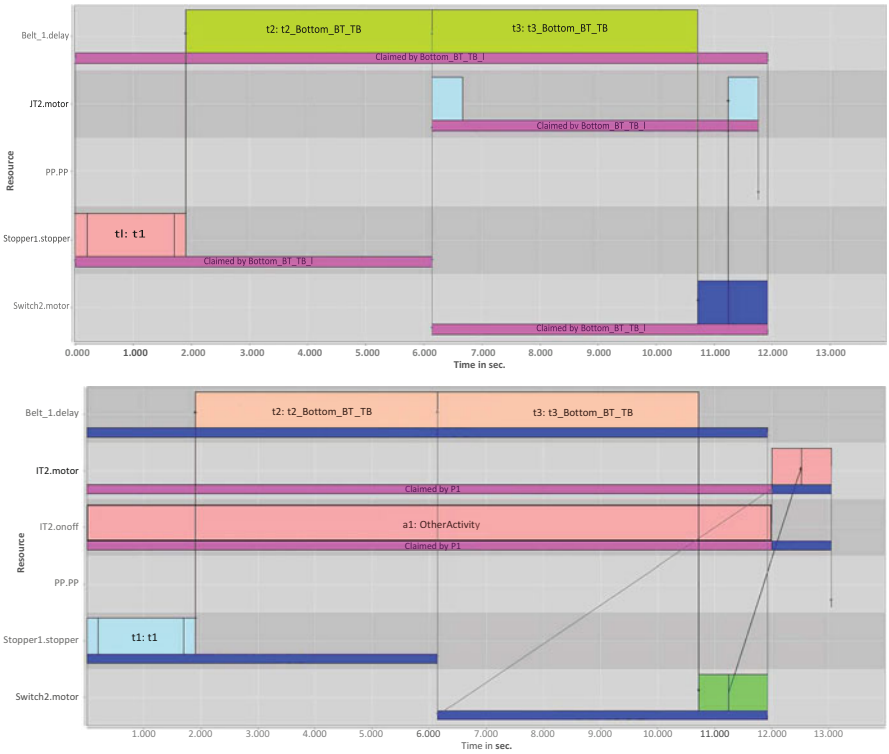


Fig. 9.6 Gantt charts for executing activity *Bottom*(1, 4.24, 4.57) of Fig. 9.4 from two initial states

execution. The maximal value in the resulting resource-availability vector gives the completion time or the makespan of the activity execution, 11.91 time units in this example, corresponding to the release of the Switch2 and Belt(1) resources.

If for some reason resource IT2 is not available at time 0 but only at time 12.00, its claim in the bottom activity is delayed until that time. The release of PP and IT2 is then delayed till 13.04, resulting in a transition from $[0, 0, 12.00, 0, 0, 0, 0, 0]^T$ into state $[6.14, 11.91, 13.04, 13.04, 0, 11.91, 0, 0]^T$. The release of the Switch2 and Belt(1) resources is not affected.

Figure 9.6 illustrates the two execution scenarios of the bottom activity discussed in this example. The charts are captured with a tool set under development for specifying, simulating, and analyzing activity models. The rows show the resource claims and action executions for the resource/peripheral pair shown on the y-axis. The second row in the charts, for instance, shows the claims and actions for the motor peripheral of indexing table IT2. Recall that peripherals were omitted from the activity models given earlier for readability purposes. The specific instance of the bottom activity considered in this example is referred to as Bottom_BT_TB_1 in the charts. The thin horizontal bars in the rows of the charts show the durations of

the resource claims. The boxes on top of those bars illustrate the action executions. Names are automatically included in those boxes if they fit and are omitted otherwise. The Stopper1.stopper row, for example, shows three action executions, corresponding to the disable, t1, and enable actions, respectively. Names follow the format "action instance:action type." In the activity models given earlier, for simplicity, we did not distinguish between instances and types because none of the action models has multiple instances of a single action. Arrows between action executions in the Gantt charts correspond to precedence relations in the activity models. The delayed availability of the IT2 resource in the bottom chart is modeled by an extra activity claiming that resource until the mentioned availability time. It can be seen from the charts that all timings are consistent with the numbers given above. Note that the PP claim and release occur instantaneously, in line with the fact that this claim/release pair only signals the arrival of a bottom piece at the PP unit.

A careful analysis of the timing of the two execution schemes discussed in this example reveals an interesting aspect of the second execution in which availability of indexing table IT2 is delayed. It can be seen that Switch2 is enabled before IT2 is properly aligned. This actually means that the bottom piece is pushed off the belt while IT2 is not ready yet, implying that the piece drops from the machine. This example illustrates that actions need to respect certain ordering and timing constraints to ensure functionally correct behavior. Correct functional behavior should be ensured by proper activity and control design. The analysis presented in this section focuses on timing only. The combination of control design and timing analysis can be used to ensure proper functionality while optimizing performance. In the considered xCPS setup, the incorrect behavior observed in the above example cannot occur given the assumptions on the input sequences. ■

The xCPS example shows two important points: (1) the completion time of an action in an FMS can be derived by considering resource-availability times, taking the *maximum* of the completion times of all preceding actions, and *adding* the action execution time; (2) the aggregated execution of all actions in an activity can be captured in terms of transitions between vectors of resource-availability times. Timing analysis of activity models can thus be done naturally in (max,+) algebra [5, 13]. The analysis framework is essentially the same as the framework discussed in Chap. 8 for dataflow models. A *resource time stamp* vector $\boldsymbol{\gamma}_R : R \rightarrow \mathbb{R}^{-\infty}$ represents the system state in terms of resource availability. For each $r \in R$, $\boldsymbol{\gamma}_R(r) \in \mathbb{R}^{-\infty}$ is the availability time of resource r. Starting from resource vector $\boldsymbol{\gamma}_R$, the execution of an activity Act leaves the system in a new state $\boldsymbol{\gamma}'_R$. This new state is determined by computing the resource release times in the activity, taking into account the dependencies and execution times of its actions, as illustrated in the above xCPS example. The essential timing information of Act can be captured in an $|R| \times |R|$ matrix $\mathbf{M}_{Act}$ that for each entry $[\mathbf{M}_{Act}]_{i,j}$ contains the duration of the longest path from resource j to resource i (or $-\infty$ if there is no path). In terms of the system scenario framework, an activity is a scenario corresponding to a set of similar run-time situations; the matrix captures the deterministic (worst-case) timing aspects of the scenario. With such a timing matrix, the vector $\boldsymbol{\gamma}'_R$ can

be computed as $\boldsymbol{\gamma}'_R = \mathbf{M}_{Act} \otimes \boldsymbol{\gamma}_R$, where $\otimes$ is the (max,+) matrix multiplication. Given $m \times p$ matrix $\mathbf{A}$ and $p \times n$ matrix $\mathbf{B}$, the elements of the resulting matrix $\mathbf{A} \otimes \mathbf{B}$ are determined by: $[A \otimes B]_{ij} = \max_{k=1}^{p}([A]_{ik} + [B]_{kj})$. The completion time or makespan of executing activity Act in state $\boldsymbol{\gamma}_R$ is $\|\boldsymbol{\gamma}'_R\|$, where for any vector $\mathbf{x}$, $\|\mathbf{x}\| = \max_i \mathbf{x}(i)$ denotes the (max,+) vector norm of $\mathbf{x}$.

xCPS Case Study 3 Consider again the $Bottom(i, d_2, d_3)$ activity of Fig. 9.4. For each instantiation of the parameters, it has a (max,+) matrix that defines the timing information. The following matrix characterizes $Bottom(1, d_2, d_3)$, for any value of the d_2 and d_3 parameters. The expressions in the matrix can be derived along the lines sketched earlier in the previous xCPS example. They can be computed by a (symbolic) traversal of the activity graph (see [14]).

	Stopper1	Switch2	IT2	PP	Turner	Belt(1)	Belt(2)	Belt(3)
Stopper1	$1.9 + d_2$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$1.9 + d_2$	$-\infty$	$-\infty$
Switch2	$3.1 + d_2 + d_3$	$d_3 + 1.2$	$-\infty$	$-\infty$	$-\infty$	$3.1 + d_2 + d_3$	$-\infty$	$-\infty$
IT2	$2.94 + d_2 + d_3$	$d_3 + 1.04$	1.04	0	$-\infty$	$2.94 + d_2 + d_3$	$-\infty$	$-\infty$
PP	$2.94 + d_2 + d_3$	$d_3 + 1.04$	1.04	0	$-\infty$	$2.94 + d_2 + d_3$	$-\infty$	$-\infty$
Turner	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$
Belt(1)	$3.1 + d_2 + d_3$	$d_3 + 1.2$	$-\infty$	$-\infty$	$-\infty$	$3.1 + d_2 + d_3$	$-\infty$	$-\infty$
Belt(2)	$-\infty$	$-\infty$	$-\infty$	∞	$-\infty$	$-\infty$	0	$-\infty$
Belt(3)	$-\infty$	$-\infty$	$-\infty$	∞	$-\infty$	$-\infty$	$-\infty$	0

It can be verified that $\mathbf{M}_{Bottom(1,4.24,4.57)} \otimes \mathbf{0}$ equals vector $\gamma = [6.14, 11.91, 11.75, 11.75, 0, 11.91, 0, 0]^T$. For example, $(\mathbf{M}_{Bottom(1,4.24,4.57)} \otimes \mathbf{0})(IT2) = max(2.94 + 4.24 + 4.57, 4.57 + 1.04, 1.04, 0, -\infty, 2.94 + 4.24 + 4.57, -\infty, -\infty) = 11.75$. This is consistent with the previous example. The makespan of executing activity $Bottom(1, 4.24, 4.57)$ in initial state $\mathbf{0}$ is $\|\boldsymbol{\gamma}\| = \max_i \gamma(i) = 11.91$. ■

xCPS Case Study 4 Similar matrices as the one presented in the previous xCPS example can be created for the *Bottom* activities using belt resources Belt(2) and Belt(3), and for the *Top*, *InvertedTop*, and *Assembly* activities. With these matrices, resource release times and makespans for these activities can be computed. An important point is to obtain the proper timing information for the actions in the activity models. Recall that the goal of the analysis is to explore productivity, covering productivity lower bounds, maximal achievable productivity, and productivity bottlenecks, during early design. For this purpose, we need tight worst-case bounds on action timing behavior.

The timing information in the activity models of xCPS case study 1 and the derived values in the matrix of xCPS case study 3 have been obtained from a set of Simulink simulations. The Simulink model is a hybrid discrete-event, continuous-time model that captures the detailed xCPS action behavior. For the purpose of this case study, it suffices to capture the behavior of the part of xCPS related to conveyer belt 2. This model includes all relevant aspects and details, such as the behavior of the turner and the dynamic behavior of the Belt 2 controller. Such a model allows to derive accurate action timing. It is important to carefully consider

the assumptions introduced in xCPS case study 1 to obtain the proper model and the correct simulations. It follows from the assumptions that the timing of processing a specific piece may be influenced by (at most) the two next pieces in the input, because three pieces share Belt 2. For instance, a bottom piece may be slowed down on Belt 2 if it is followed by an inverted top piece that needs to be flipped in the turner, whereas it is not slowed down if it is followed by a regular top. Also at most two pieces preceding the modeled piece in the input may have an impact on the timing. This is because those pieces may or may not have been flipped, which leads to different Belt 2 controller timing at the moment the current piece is being processed. Thus, the timing of processing a piece depends on a context of two preceding and two following pieces. These timing variations are captured in the parameters d_2, d_3, and d_4. Figure 9.7 shows timing diagrams of some of the performed simulations. The figure shows three time–displacement profiles for pieces on Belt 2, in line with the assumption that three pieces can share this belt. The simulated sequence starts with a bottom piece, followed by an inverted top, another bottom, and a regular top. Precise timing and position information can be obtained from those diagrams. A bottom piece, for instance, leaves Belt 2 at Switch 2 at position 27; a top piece leaves at position 35. Both the fast and the slow movement

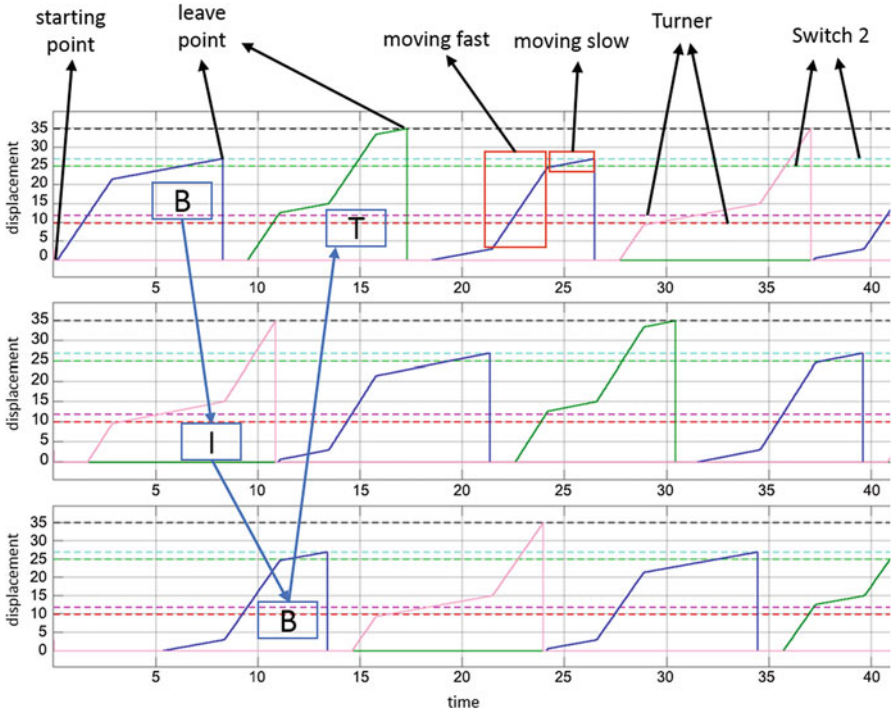


Fig. 9.7 Measurements for action timing information. The *y*-axis shows the displacement of pieces on Belt 2. The *x*-axis shows progress of time

are visible. It can be observed that the processing of a bottom depends on its context of (inverted) top pieces.

The matrices for the xCPS activity models have been obtained from a sufficiently large set of simulations. The action timings in the models are the measured worst-case values. Note that this does not guarantee that the actual (theoretical) worst-case behavior has been observed. The detailed dynamics of all the actions in a complex machine like xCPS is difficult to analyze. Simulations suffice though to obtain values suitable for early design phase analysis and exploration. ■

The introduced xCPS activity model is a (max,+) switched linear system, as introduced in Sect. 3.4 of Chap. 8. It is straightforward to verify that it is both additive/monotonic and homogeneous/shift-invariant, as illustrated also by the examples in xCPS case study 2. So the analysis techniques and properties of (max,+) linear systems discussed in the previous chapter apply to the model.

Besides requiring that the activity model satisfies the conditions of a linear system, it needs to be ensured that the model is a sound abstraction of the detailed behavior of the FMS under consideration in terms of the abstraction–refinement relation discussed in Sect. 2.2 of Chap. 8. That is, the model should correctly capture upper bounds on the completion times of actions, relative to the availability times of the required resources. This is ensured when the measured worst-case action execution times may be assumed to be proper conservative bounds on action timing, as explained in xCPS case study 4. Note that the model encodes context information about the sequence of pieces to allow for a *tight* abstraction of the behavior modeled as an activity.

4.3 Analyzing Activity Sequences

We have seen how to model and analyze the timing behavior of a single activity execution. System-level timing behavior can now be analyzed by considering possible sequences of activities. Figure 9.8 shows part of the Gantt chart corresponding to the execution of an alternating sequence of bottom and top pieces in xCPS. We would like to know timing aspects such as the makespan of a finite sequence of processed pieces, or the guaranteed throughput that can be achieved by such a sequence in steady state.

Using the (max,+) framework introduced in the previous subsection, the resource-availability times of the execution of a sequence $Act_1 Act_2 \cdots Act_n$ of activities starting from a vector $\boldsymbol{\gamma}_R$ of resource-availability times is given by expression $M_{Act_n} \otimes \cdots \otimes M_{Act_2} \otimes M_{Act_1} \otimes \boldsymbol{\gamma}_R$. The makespan can be obtained from these resource-availability times, in the form of the maximum value in the vector.

xCPS Case Study 5 Let us consider execution of scenarios *Bottom*(i,4.24,4.57) and *Top*(i,1.64,4.71,1.00) pieces followed by an *Assembly* activity, denoted B, T, and A for short. Consider the sequence *BTABTA* starting from the initial state $\mathbf{0}$. This sequence leads to resource-availability times [19.38, 21.61, 21.45, 27.38, 19.38,

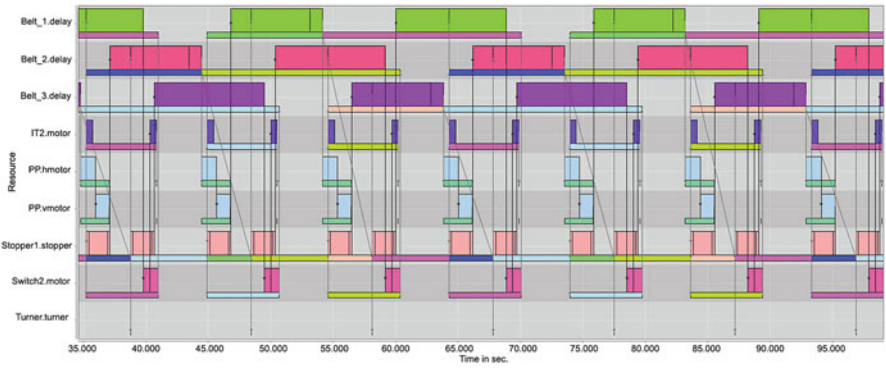


Fig. 9.8 Gantt chart of a steady-state repetitive *BTA* execution

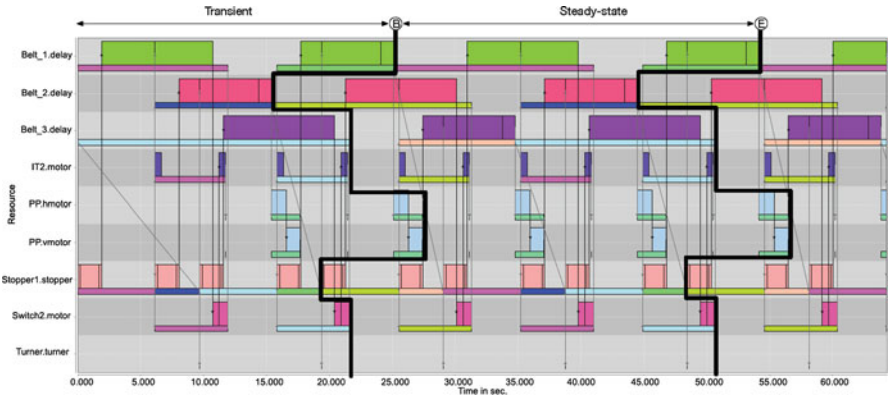


Fig. 9.9 Gantt chart of a repetitive *BTA* execution from the initial state

25.08, 15.40, 21.61]^T and makespan 27.38. Figure 9.9 shows the Gantt chart for this sequence. The vector of resource-availability times is marked by a solid line, labeled B. We see that the pieces iteratively claim the Belt(1), Belt(2), and Belt(3) resources, conforming to the assumption that at most three pieces are on the belt simultaneously. ■

Besides the makespan of activity sequences, we are also interested in steady-state productivity of an FMS. Essentially, steady-state productivity can be determined in the activity modeling framework by looking for a recurrent state in the resource availability while executing activities. Since time increases, resource-availability vectors need to be normalized to find recurrent states. For vector $\mathbf{x}$, with $\|\mathbf{x}\| > -\infty$, $\mathbf{x}^{\mathbf{norm}}$ equals $\mathbf{x} - \|\mathbf{x}\|$, the normalized vector, with $\|\mathbf{x}^{\mathbf{norm}}\| = 0$. That is, resource-availability vectors are normalized by subtracting the maximum entry from all its elements. Productivity, or throughput, of a sequence of activity executions can then be computed by considering the activity subsequence repeating between recurrent

states. The output produced by this repeating subsequence should be divided by the makespan of this sequence. Such a recurrent state always exists, because executing a finite number of activities allows only a finite number of normalized states [15, 42]. In practice, throughput of a repetitive activity sequence is not computed by exploring the state space corresponding to the sequence. It can be computed more efficiently by computing the largest eigenvalue of the product of the matrices of the repeated activities. This eigenvalue corresponds to the period of the repetition, from which the throughput can be derived straightforwardly.

xCPS Case Study 6 Consider again Fig. 9.9. The normalized resource-availability vector after executing the initial *Bottom*(1,4.24,4.57) activity is $[6.14, 11.91, 11.75, 11.75, 0, 11.91, 0, 0]^T - 11.91 = [-5.77, 0, -0.16, -0.16, -11.91, 0, -11.91, -11.91]^T$. The normalized state after executing the *BTABTA* sequence from the initial state (see also xCPS case study 5) is $[-5.76, 0, -0.16, -0.16, -11.91, -6.21, 0, -9.68]^T$. This turns out to be a recurrent state, which repeats every time sequence *BTABTABTA* is executed. The recurrent normalized resource-availability vector after executing sequence $(BTA)^5$ is marked with the solid line E in Fig. 9.9. The repeated sequence *BTABTABTA* has a makespan of 29.13 s when starting from this recurrent vector of resource availability and produces three composite pieces. This makespan corresponds to the largest eigenvalue of the matrix for sequence *BTABTABTA*. This leads to a throughput of approximately 0.103 composite pieces per second, because three composite pieces are produced in one repetition. ■

This example considers the throughput of a single execution pattern. In general, we would like to know the timing aspects of all possible behaviors. To this end, all possible activity sequences need to be specified in a finite-state machine. Such a state machine can then be translated to a state space of normalized resource-availability vectors, essentially following the above example. Any cycle in this state space corresponds to a possible steady-state behavior of the modeled system. Any such cycle also corresponds to a throughput value. A lower bound on the throughput of the system is given by the minimal throughput value among all cycles; this throughput can always be guaranteed. The best achievable throughput is obtained from the maximal throughput value among all cycles in the state space. Any cycle with maximal throughput value also provides an activity ordering that realizes this throughput. Such an ordering can be used as schedule or control strategy if the occurrence of the activities can be controlled. The next section discusses this so-called supervisory control in more detail. Note that the action execution times underlying the (max,+) matrices of the activities are assumed to be worst-case execution times. This implies that in reality an activity sequence with maximal throughput may show an even higher throughput than the concrete value that the best-case analysis shows. This concrete value, though, is the throughput that can be guaranteed for such an activity sequence under all possible action execution times. The details of the throughput analysis are beyond the scope of this chapter; the interested reader is referred to [15, 16, 42] and the previous chapter. In practice, the explicit construction of the normalized state space can be avoided for the throughput lower bound. This lower bound can be computed on the often smaller (max,+)

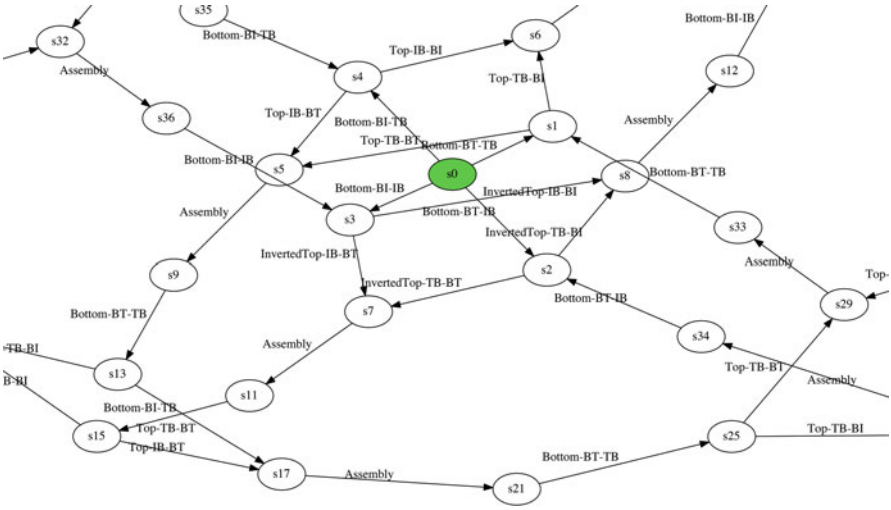


Fig. 9.10 A fragment of the FSM capturing all possible xCPS activity orderings

automaton graph, as explained in Sect. 3.3 of Chap. 8. The explicit computation of the state space cannot be avoided for the best-case analysis.

xCPS Case Study 7 Figure 9.10 shows a fragment of the FSM that captures all possible activity sequences of xCPS in line with the assumptions of xCPS case study 1. Transition labels provide the executed activity and its context, as explained in xCPS case study 4. For instance, transition Bottom-BT-TB from initial state s0 to state s1 corresponds to activity *Bottom*(1,4.24,4.57) that we have considered in earlier examples; the context BT-TB specifies that the bottom piece is preceded by a *BT* combination and followed by a *TB* combination (implying the value of the transport delay values $d_2 = 4.24$ and $d_3 = 4.57$). The repetitive sequence of *BT* combinations analyzed in the earlier examples corresponds to the state sequence s0(s1s5s9s13s17s21s25s29s33)$^\omega$ in the FSM, where x^ω denotes infinite repetition of x.

The FSM of Fig. 9.10 together with the activity models completes the model for xCPS. It can be analyzed for various timing properties using existing (max,+)-based techniques for activity and dataflow models, as explained earlier. The repetitive sequence of *BTA* activities corresponds to the best possible throughput of 0.103 composite pieces per second. As explained above, the actual throughput may be higher if the actions in the activities do not run with their worst-case execution times, but the 0.103 pieces per second can be guaranteed independent of the low-level action timing. The throughput lower bound among all possible activity sequences occurs when all tops in the input are inverted. The throughput is then 0.080 pieces per second. Since this is a lower bound, it holds for all action timings. Since, for now, the orientation of top pieces in the input is assumed to be uncontrollable, the analysis shows that the productivity of xCPS is between 0.080 and 0.103 pieces per

second, with potentially (small) positive outliers if actions do not show worst-case timing.

The scenario-based analysis using activity models is fast. The throughput analysis giving the mentioned best-case and worst-case throughput values, for instance, takes 8.85 ms. This is much faster than traditional simulation would be. The activity-based analysis is moreover exact, under the assumption that the action execution time values are tight, worst-case values. All possible behaviors are taken into account. This is typically infeasible, or very time consuming, with traditional simulation. Note that, since action execution times are worst-case values, the actually observed throughput might in reality be slightly higher than the mentioned best-case throughput value of 0.103 composite pieces per second. However, the lower-bound value of 0.080 is a guaranteed value below which the throughput will never drop. ■

4.4 Bottleneck Identification

By analyzing the timing behavior of activity sequences, we can determine system-level makespan and throughput guarantees. Performance improvements can be achieved by resolving productivity bottlenecks. In the context of activity models, these can be identified using the critical path method (CPM) [21]. The CPM identifies the critical path(s) in a given activity sequence. A critical path is defined as a longest path of peripheral actions in the activity sequence for which a delay on their start or completion times would affect the overall makespan. Resolving the bottlenecks, for instance, by improving the execution times of critical actions, leads to an increase in the overall performance. In the context of design, this translates to an upgrading or replacing of certain system peripherals or resources.

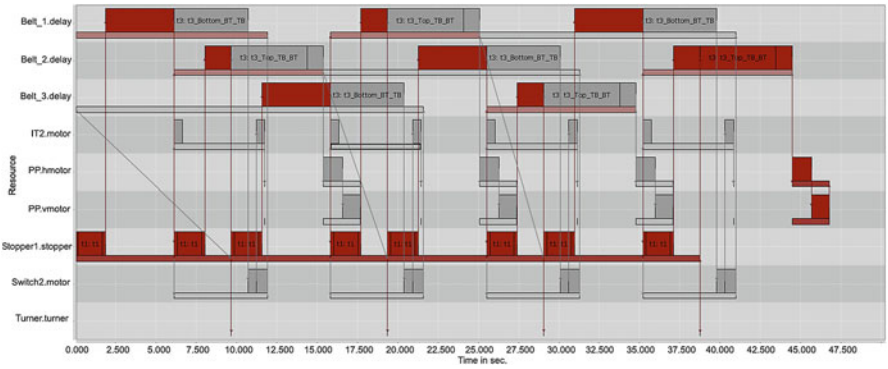


Fig. 9.11 Critical path of the *BTABTABTABTA* sequence

xCPS Case Study 8 Figure 9.11 depicts a Gantt chart highlighting the critical path of the xCPS activity sequence *BTABTABTABTA*. Critical actions are depicted in red, while other actions are depicted in gray. We may conclude that resources Stopper1 and Belt(1), Belt(2), and Belt(3) (which together model conveyer belts 2 and 3) are currently the productivity bottlenecks. Updating either of these resources to make them faster would lead to better system performance. The speed of Stopper 1 is physically limited, since reducing the duration of t1 would make the stopper hit passing pieces. Thus we are left with the option to improve the belt resource. Section 6 explores the design of different controllers for conveyer belt 2, namely a standard feedback controller running on a single processing core, a pipelined controller running on multiple cores of a multi-core platform, and a scenario-based pipelined controller that switches between different multi-core configurations. The better a controller is, the faster the belt reaches its desired speed set point. This has an impact on the duration of the transport actions t1–t4 in the activity models. Thus, controller performance may have an impact on system-level performance. The productivity numbers presented in xCPS case study 7 are obtained using the pipelined multi-core controller. Using the best controller obtained in Sect. 6, i.e., the scenario-based controller, the guaranteed productivity can be improved to 0.111 pieces per second, which is an improvement of 7.2% compared to the 0.103 pieces per second obtained with the non-reconfigurable pipelined controller. The analysis uses adapted activity models with updated t1–t4 durations, and then goes along the same lines as explained in this section. This analysis is fully automated and hence very fast.

A different design exercise may focus on cost reduction instead of performance improvements. The cost reduction should come without hindering system performance. To this goal, we can play with resources that are not in the critical path, for instance, indexing table IT2. Figure 9.12 depicts the critical path of the same activity sequence as before, but with IT2 virtually replaced by a slower candidate indexing table resource cIT2 with actions that take six times longer than those of IT2. The identified bottlenecks remain the same and the makespan is unchanged.

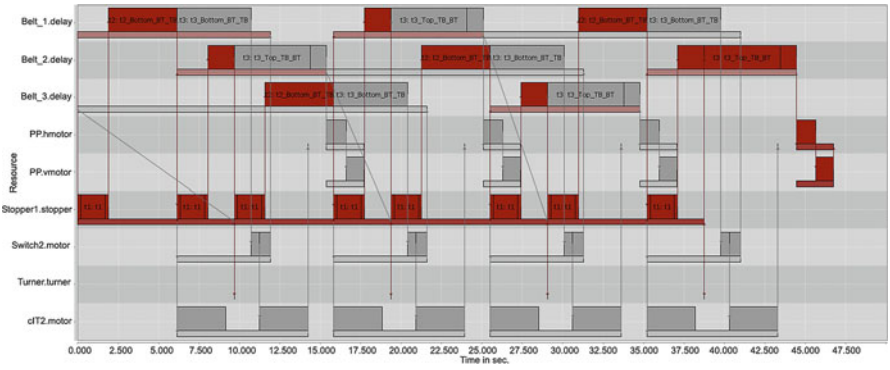


Fig. 9.12 Critical path of the *BTABTABTABTA* sequence assuming a slower IT2 resources

This shows that we can use slower (and in principle cheaper) resources to obtain the same performance. ■

In conclusion, activity models enable efficient system-level performance analysis, bottleneck identification, and design-space exploration for specific input/activity sequences, captured in an FSM. The next section discusses techniques to create such an FSM of acceptable behaviors and optimize it for performance. Combining these techniques with bottleneck identification allows for a systematic approach to narrow down design choices that improve the system performance, first by determining the appropriate activity orderings and then by identifying the bottlenecks that can be further improved.

5 Synthesis and Optimization of Supervisory Controllers

As illustrated in Fig. 9.10, a finite-state machine can be used to specify all allowed activity sequences. Constructing such an FSM manually becomes infeasible when the number of activities and activity ordering constraints gets large. An alternative approach is to specify the possible behaviors in a modular fashion. Properties of each part of the system can then be modeled in isolation, and the composition of the models forms the complete system model. Restricting a system model to ensure that only proper behavior is allowed is referred to as *supervisory controller synthesis*. The resulting FSM is referred to as a (supervisory) controller, or supervisor. This section presents a scenario-based controller synthesis approach based on activity models, following [42]. The approach facilitates the specification and modeling of complex FMS behavior. It moreover improves efficiency and scalability of controller synthesis and leads to more compact controllers, when compared to an approach that models FMS behavior in terms of low-level actions. Furthermore, the approach allows performance optimization of the resulting controller. Finally, the approach supports code generation, allowing to automatically obtain controller implementations from the generated controller model. Section 5.1 presents our modeling and controller synthesis approach in more detail. Section 5.2 discusses makespan and throughput optimization. Section 5.3 explains code generation.

5.1 Modeling and Controller Synthesis

Our supervisory controller synthesis starts from a modular behavioral specification of the targeted system [41]. The modular specification captures possible activity orderings of the system, referred to as the *plant* models, and activity ordering constraints, the *requirement* models. They are captured modularly and concisely using a network of extended finite-state automata [35]. An extended finite-state automaton is an augmentation of a finite-state automaton with state variables that

can be used in guard formulas and can be updated on transitions. Each transition captures the execution of an activity. The set of activities is partitioned into a set of *controllable* activities and a set of *uncontrollable* activities. Controllable activities can be enforced by the supervisor (e.g., actuator commands), while uncontrollable activities cannot be activated or inhibited by the supervisor (e.g., changing sensor values). The composition of plant and requirement automata is characterized using multiparty synchronization [10], where execution of shared activities among different automata is synchronous. This synchronization mechanism ensures that an activity can only be executed if all involved automata are able to execute the activity at the same time; activity execution then involves all automata in which the activity is enabled, moving all of them to their specified next states.

Given the complete system behavior in the form of the composition of plant and requirement models, the Ramadge–Wonham framework of supervisory controller synthesis [32, 33], in particular its extension to extended finite-state automata [29], can be used to obtain a supervisory controller. Such a controller enforces restrictions in activity orderings upon the system to ensure safe behavior (e.g., avoiding collisions between robot arms and preventing buffer overflows), and guarantees deadlock-free behavior of the system. Furthermore, it is ensured that the controller is controllable in the sense that it does not block uncontrollable activities that can occur in the plant. Additionally, the controller is minimally restrictive, which means that the controller does not restrict the behavior of the FMS more than strictly necessary with respect to the requirements. Minimal restrictiveness allows for controller optimization as a subsequent step. The timing information of each activity, for example, can be used to find the performance-optimal control choice in each system state, considering, e.g., makespan or productivity.

Our scenario-based supervisory controller synthesis approach combines a modeling and composition step with the classical Ramadge–Wonham framework. This facilitates modeling and improves scalability of the synthesis [42]. With proper models at the abstraction level of activities, many unsafe behaviors can already be excluded in the composition step and fine-grained action-level behaviors can be avoided. This two-step synthesis approach scales better than traditional one-step synthesis, because it reduces peak memory usage. Both the composition and the post-composition synthesis typically require less memory than the one-step synthesis on a fully expanded (action-level) model that may contain many fine-grained, possibly unsafe behaviors. Peak memory usage, rather than run-time, is often the bottleneck in classical synthesis. Moreover, the state space of the generated supervisory controller is typically much smaller compared to a controller generated from an action-level model.

xCPS Case Study 9 Recall the FSM of Fig. 9.10 that models all allowed activity sequences in xCPS given the assumptions made in xCPS case study 1. Manually designing such a controller is a tedious task, and prone to errors. Therefore, we model the system using a network of plant and requirement automata, from which this controller can be generated. In the modular specification, each automaton captures a single aspect.

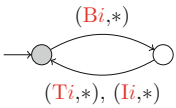


Fig. 9.13 Plant that captures the strict alternation between bottom and top pieces, starting with a bottom piece

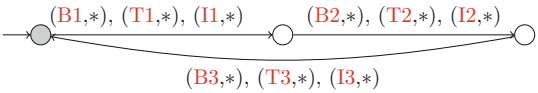


Fig. 9.14 Plant that models the cyclic ordering in claiming the Belt(i) resources

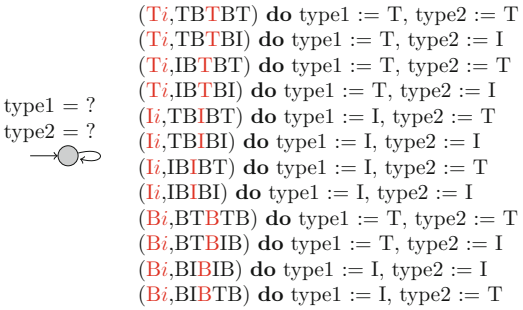


Fig. 9.15 ObserverTops: observer to keep track of the context related to the top pieces

Figure 9.13 shows the plant model that captures strict alternation between bottom and top pieces, starting with a bottom piece. For conciseness, InvertedTop is abbreviated to I. Transitions contain activity-context pairs, covering the same information as the transitions in the FSM in Fig. 9.10, but in a more compact format. Index i is used to capture which of the three resources Belt(i) is used. Each activity instance in the automata with index i should be expanded to the three instances with $i = 1$, $i = 2$, and $i = 3$. The model of Fig. 9.13 allows arbitrary contexts for the bottom and top pieces, indicated by a $*$. For example, (B1,$*$) should be expanded to activity B1 with all possible contexts. Recall that the context determines the value of the d_2–d_4 parameters in the activity models. The orientation of the top pieces that arrive is uncontrollable, which means that (Ti,$*$) and (Ii,$*$) are uncontrollable activities. Activities (Bi,$*$) are controllable activities (although only the bottom activity with the right context can be chosen, as explained below). Initial states of automata are marked with an incoming arrow. States can be marked, indicated with a gray node. Marked states can be used in modeling and synthesis to indicate system states that should always be reachable, for example, to bring the system to a safe state, or the completion of (sub-)tasks. The initial state of the automaton in Fig. 9.13 is marked to indicate that bottom and top pieces always need to be processed in pairs.

As described before, the three belt resources are claimed in order in a cyclic manner. This is captured in the plant model shown in Fig. 9.14. Figures 9.15 and 9.16 show the plant models that guarantee the correct context information for each activity. The ObserverTops plant model shown in Fig. 9.15 stores the information about the current type of top piece in variable type1, and the future

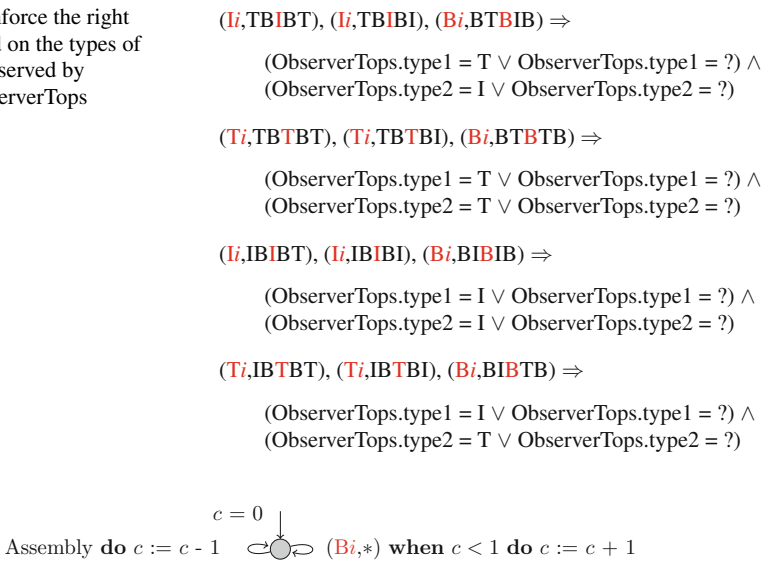


Fig. 9.16 Enforce the right context based on the types of top pieces observed by observer ObserverTops

Fig. 9.17 Requirement IndexTableBuffer to avoid a buffer overflow on the indexing table

type of top piece in variable type2. Initially the context information about the top pieces is not available, indicated by a question mark. Given this information, we can formulate a plant model, shown in Fig. 9.16, that enforces that an activity can only be executed with the correct context information. This model uses a state-based expression [23] that states the condition under which activities are allowed to occur; activity-context pairs $(A_1, C_1), \ldots, (A_n, C_n)$ are only allowed to occur if condition c is satisfied, denoted as $(A_1, C_1), \ldots, (A_n, C_n) \Rightarrow c$. Such a state-based expression is simply a textual shorthand for an automaton with a self-loop that contains the activities and the guards. It eases specification of behavior.

To ensure a correct and safe assembly process, we need to enforce various requirements. First, we need to prevent overflowing the indexing table or the belt in front of the pick and place unit with too many pieces. Considering that at most three pieces are allowed on Belts 2 and 3, we restrict the number of top pieces in front of the PP unit to three. We also limit the number of bottom pieces on IT2 to one. In line with the assumptions in xCPS case study 1, the bottom activity first aligns IT2 with Belt 2, and after the bottom piece is pushed onto IT2, IT2 is turned directly towards the PP unit. This behavior effectively limits the capacity of IT2 to one. Turning IT2 for another 90° before assembly would allow an unassembled bottom to be moved towards the output, which is undesirable. Figure 9.17 shows the requirement that at most one bottom can be inputted, and then first an assembly operation must be performed before a next bottom piece can be inputted. Figure 9.18 shows the buffering requirement for the belt capturing that at most three top pieces can be inputted. To input a next piece, first an assembly operation should be performed.

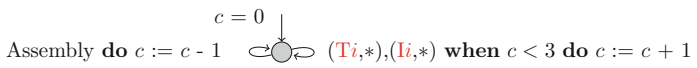


Fig. 9.18 Requirement BeltBuffer to avoid a buffer overflow on the belt

Assembly $\Rightarrow$

IndexTableBuffer.c > 0 and BeltBuffer.c > 0

Fig. 9.19 Enforce that assembly can only happen if there are bottom and top pieces to be assembled

As a final requirement, observe that an assembly can only be performed if there is both a top and bottom piece available, shown in the model of Fig. 9.19. The combination of the three requirements provides the boundaries for a correct assembly within the given assumptions.

Our two-step supervisory controller synthesis can be applied to the set of plant and requirement automata described above. The composition step already yields the same controller as is shown in Fig. 9.10; the classical Ramadge–Wonham synthesis step does not change the result anymore. Both the composition and the Ramadge–Wonham synthesis complete in a few seconds. ■

5.2 Throughput and Makespan Optimization

Due to the minimal restrictiveness of generated controllers, there may be room left for performance optimization. As already explained, an important metric for FMS that continuously repeat the same process is *productivity* or *throughput*, which quantifies the number of products that are finished per time unit. Finding a throughput-optimal control strategy is not trivial, because in general not all activities in the FMS are controllable. In the xCPS case study, for example, the orientation of the top pieces is not controllable. The throughput optimization problem for supervisory controllers with uncontrollable activities can be formalized in a game-theoretic setting. The system controller is one player in the game, and the environment (that determines uncontrollable activities) is its adversary. A controller strategy is needed that maximizes the ratio of productivity divided by delay in the resource-availability vector state space, given environmental influences on uncontrollable activities. Such a controller strategy against uncontrollable activities corresponds to finding an optimal strategy in a ratio game [40]. When the system is fully controllable, i.e., when we have full control over all activity orderings, the strategy reduces to a repeatable activity sequence in the controller that yields the highest throughput. To find such a sequence, we can simply apply the technique

explained in Sect. 4.3 and the previous chapter, using the state space of normalized resource-availability vectors generated by the controller.

xCPS Case Study 10 Consider again the controller of Fig. 9.10. The worst- and best-case guaranteed throughput of the system have already been described in xCPS case study 7. A throughput lower bound of 0.080 composite pieces per second occurs when all tops enter the system in inverted orientation. Since the controller does not have different control options in response to an inverted top entering the system, the mentioned game-theoretic analysis, trivially, confirms this result as the best achievable lower bound on throughput with uncontrollable top orientation. If we assume that all activities are controllable, i.e., if it is possible to control the orientation of the tops, then the results of the throughput analysis presented in the previous section yield that the repetitive sequence of BTA activities corresponds to the best possible throughput of 0.103 composite pieces per second.

If we consider a variant of the xCPS case study in which also the input order of pieces is controllable, then the controller has many more options. With the given timing, however, the Belt(i) and Stopper 1 resources are the bottleneck (see Sect. 4.4). A repetitive sequence of BTA activities is then still optimal.

However, let us consider an xCPS variant with a slower index table IT2 such that IT2 becomes the bottleneck. Recall the assumption made in xCPS case study 1 that IT2 rotations are restricted to 90° rotations, with the consequence already mentioned in the previous subsection that the IT2 capacity is effectively limited to one. A more fine-grained control of IT2, where we use multiples of 30° rotations (see also the xCPS layout of Fig. 9.3), allows a more effective use of IT2, increasing its buffering capacity to two. Throughput can then be optimized by ensuring that no buffer place on IT2 is left empty.

To perform this analysis, we need adapted activity models. The IT2 align belt and align pp actions are removed from the Bottom activity. Instead, we introduce three new activities, with only a single action, to model 30°, 60°, and 90° IT2 rotations: Turn30, Turn60, and Turn90, abbreviated 30, 60, and 90 in activity sequence expressions. We assume that these movements take 6.0s, 12.0s, and 18.0s, respectively.

We also need updated and additional plant and requirement models. The plant model of Fig. 9.13 capturing the strict alternation of bottom and top pieces needs to be removed from the specification. The buffer capacity constraint in the automaton of Fig. 9.17 is set to two. The alignment of IT2 with the belt of the PP unit is observed using the plant model shown in Fig. 9.20. For pushing a bottom piece onto IT2, and assembling a top and a bottom piece, the correct alignment is needed, captured in the requirement automaton of Fig. 9.21. The requirement automaton in Fig. 9.22 enforces that IT2 turns before a next bottom piece is inputted. Only one

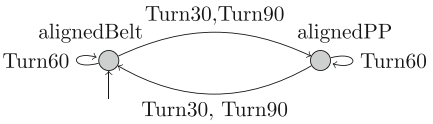


Fig. 9.20 Plant that captures the alignment of IT2 with the belt or pick and place unit

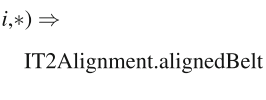


Fig. 9.21 Enforce a correct alignment of IT2 for pushing a bottom piece onto it, and for assembly

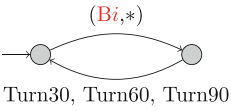


Fig. 9.22 Enforce that IT2 turns before a next bottom piece is inputted

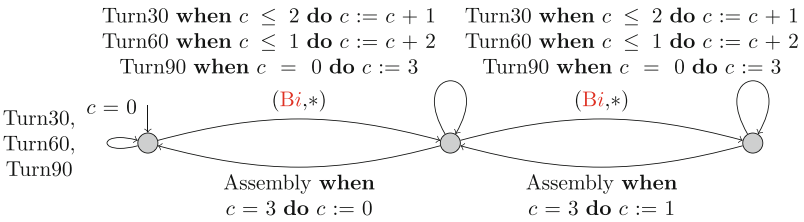


Fig. 9.23 Enforce that an unassembled bottom piece does not pass the pick and place unit

turn operation is allowed, and we assume that initially the indexing table is aligned with the belt. Finally, we need to guarantee that an unassembled bottom piece does not pass the PP unit as shown in Fig. 9.23. Counter c denotes how far the one or two bottom pieces on IT2 have moved, in number of 30° rotations. To ensure that the PP unit is not passed, this value cannot become greater than 3. After assembly, the counter is reset if there is no bottom remaining on IT2, and otherwise, the next bottom to be assembled must still need a turn of 30°, so c is set to 2.

From the set of models obtained, we can now generate a supervisory controller, which contains 603 states and 1787 transitions. The mentioned game-theoretic throughput analysis yields as optimal sequence the sequence B60(BT30A30)$^\omega$ with a throughput of 0.067 composite pieces per second, whereas (B90TA30)$^\omega$ corresponding to the alternating processing of bottom and top pieces, which was optimal so far, has a throughput of 0.037 composite pieces per second. ■

Makespan optimization is used in batch-oriented FMS to find the activity sequence allowed by a supervisory controller that minimizes the total production time. Given an initial state and a set of final states in a controller, finding the minimum makespan reduces to a shortest-path problem in the generated resource-availability-vector state space, assuming transitions in this space are labeled with the activity delay (captured by the vector norm used for normalizing the states). There are efficient graph algorithms to solve this problem, under the assumption that the considered graph has no negative-weight cycles. In our case, this assumption holds since each activity delay is non-negative. One well-known algorithm for finding shortest paths is Dijkstra's algorithm [12], which has quadratic time complexity in the number of states.

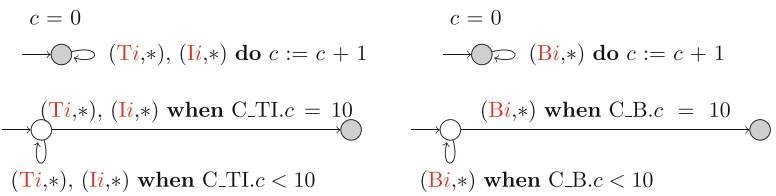


Fig. 9.24 Plant models that capture that ten tops and ten bottoms are inputted

xCPS Case Study 11 Consider again the xCPS variant with a slow IT2, fine-grained IT2 control, and controllable input order. To model the production of a batch of ten composite pieces, the specifications given so far in this section for this variant can simply be extended with the four plant models shown in Fig. 9.24 that express that ten bottoms and ten tops, in any orientation, are inputted in arbitrary order. A final state in the system is reached when each automaton has reached a marked state, indicated with a gray colored node in the automata pictures. In the batch model under consideration this happens when 10 tops and 10 bottoms have been inputted.

A supervisory controller containing 1403 states and 3943 transitions can be obtained by applying synthesis on the resulting model. Note that this controller is larger than the controller for continuous processing discussed in the previous xCPS case study example, because no recurrent state exists when processing finite batches. The makespan of the possible input sequences can be analyzed in the resource-availability state space. A shortest-path algorithm yields the optimal sequence $\text{B60(BT30A30)}^9\text{T30A}$ with a makespan value of 164.92 s, whereas the sequence $(\text{B90TA30})^{10}$ has a makespan of 278.92. Note that the schedule makes use of the optimal periodic strategy obtained earlier with appropriate pre- and postamble. ■

The timed resource-availability state space of a generated supervisory controller may become very large, despite the modular, two-step, scenario-based approach. Scalability of the performance optimization presented in this subsection may improve if we can restrict this state space. The state space may in fact exhibit redundant interleaving of activities with respect to functionality and performance aspects like throughput. The relative ordering of activities that do not share resources has no impact on timing behavior. This may lead to many controller execution paths that are equivalent from a functionality and performance point of view. Using a partial-order reduction technique [40, 43], we can compute a reduced controller without redundant paths directly from the plant and requirement models. This smaller timed state space allows faster performance analysis. In the xCPS models developed in this chapter, all activities share resources or are dependent because they occur in the same automaton, so the state space has no redundancy. However, for other cases, depending on the amount of redundancy in the models, reductions in state and transition counts up to 80.4% and 91.4% for a wafer logistics model of a lithography scanner can be obtained [40].

An alternative approach to improve scalability is based on the so-called constraint automata [8] that capture constraints with respect to aspects such as input

orderings, capacities, and safety. Using a dedicated constraint operator to compose these constraint automata with (other) requirement and plant automata prunes the resulting state space during composition.

5.3 Code Generation

A synthesized supervisory controller that has been optimized to find optimal control choices in each system state still needs to be implemented in the actual system. Such a controller typically needs to execute under real-time constraints to ensure timely responses to events in the system. The generation of implementation code for supervisory controllers can be automated if the control API of the target system is known. An optimized controller has in each state either one controllable activity or (possibly multiple) uncontrollable activities enabled. In a state with one or more uncontrollable activities, the controller has to wait till an uncontrollable activity occurs, resolving the choice in case multiple activities are enabled. The controller maintains a schedule of all actions that have been executed, are currently executing, or still have to be executed. This schedule is executed on-the-fly, and the actions of new activities can be added to this schedule in real-time. Actions of activities that have been fully executed can be removed from the schedule, to keep the run-time memory usage limited.

xCPS Case Study 12 We have implemented code generation for xCPS. Each action in an activity is mapped to a hardware function of the xCPS API. The worst-case timing abstraction that is assumed in the activity models is replaced by fixed timing delays or sensor feedback to indicate that an action has finished. For example, consider Fig. 9.25, where we assume a fixed timing delay to open (disable) Stopper 1. Figure 9.26 shows a code fragment of the controller code. An uncontrollable choice occurs for the type of top piece that is being inputted. In state s0, we schedule activity Bottom, and then add a callback to Sensor 2 for state s0_type. When a top piece arrives at Stopper 1, Sensor 2 is triggered (see Fig. 9.3), and the type of the next piece can be retrieved in state s0_type. Based on the type, the activity is executed, and the next state is reached. ■

```
bool Controller::disable_stopper1(API &api,::xCPS::xCPS &xcps,const Vertex &v){
  xcps.get_component<Stopper>("stopper1").disable();
  add_callback(convertToCallback(this, &Controller::notifyReady,v),1,
    DISABLE_DELAY);
  return false;
}
```

Fig. 9.25 xCPS action to disable Stopper 1. DISABLE_DELAY is the time delay needed before Stopper 1 is fully opened

```
enum Type = {T,I,B};
bool Controller::s0(){
  scheduleActivity("B");
  add_sensor_callback("sensor2",createSensorCallback(this,&Controller::s0_type));
}
bool Controller:s0_type(){
  Type type = get_sensor_value("sensor2");
  if(type == Type.T) {
    scheduleActivity("T");
    s1();
  } else if(type == Type.I) {
    scheduleActivity("I");
    s2();
  }
}
```

Fig. 9.26 Part of the xCPS controller

6 Scenario-Based Data-Intensive Feedback Control

The previous sections focused on scenario-based performance optimization and supervisory control of FMS. In this approach, scenarios correspond to activities. Activities capture deterministic system functionality with tightly bounded action timing. The implementation of actions and activities is typically achieved via various types of feedforward and feedback control. With the increasing availability of affordable, powerful multi-core embedded platforms, compute-intensive image-based control (IBC) is gaining interest. In an IBC system, a camera and an image processing algorithm are used in combination as a sensor. Processing images in a control loop introduces a relatively long sensing delay in the loop due to the processing latency. On a multi-core platform, however, the image processing can be parallelized and/or pipelined. If subsequent images can be processed independently, from an implementation perspective, pipelining is relatively straightforward. Pipelining the sensing in IBC loops does not reduce the sampling latency, but it does increase the sampling rate of the IBC system. This in turn may improve quality of control (QoC). A static pipelining configuration that allocates a fixed number of cores to the pipelined sensing may be expensive in terms of processing resource usage though. A reconfigurable, scenario-based pipelining approach improves resource efficiency when an embedded platform is shared with sporadic tasks (such as a supervisory controller or other controllers needed to implement actions). A pipelining configuration of the IBC system is then a system scenario of the IBC system in the scenario terminology of Chap. 2. This use of the scenario approach is independent of the scenario-based modeling explained in the earlier sections. It is also applicable to other domains, like advanced driver assistance systems (ADAS). In the FMS context, the scenario-based performance analysis can be used though to analyze the effect of improved QoC of a scenario-based pipelined IBC system on FMS productivity.

This section first introduces IBC in Sect. 6.1. Section 6.2 explains how to define system scenarios in IBC. Section 6.3 presents our scenario-based pipelined control (SBPC) approach.

6.1 Image-Based Control

IBC is an example of data-intensive control, where large amounts of data are being processed as part of a controller. The use of high-resolution images allows to sense physical *states* of an FMS that are hard to measure by other technologies. Traditional implementation of an IBC system performs *sensing* (sensing the system states), *computing* (computation of the control signal), and *actuating* (application of the control signal to the system) operations (or tasks) sequentially as shown in Fig. 9.27a. Such sequential implementation results in a long sampling period (i.e., the time interval between the end of two consecutive actuation tasks) and consequently, a degraded control performance, or QoC (e.g., speed of the response) [39].

Pipelined control is an implementation alternative to address this problem [26]. Pipelined control implements the sensing algorithm in a pipelined fashion using parallel processing resources as illustrated in Fig. 9.27b. In essence, such pipelined implementation reduces the sampling period by actuating more frequently than the sensing period. This opens up the possibility for QoC improvement by appropriate controller design techniques. Generally, a higher number of processing resources leads to a shorter sampling period and a higher QoC. Processing resources, however, may be shared with other tasks and may not be always available. Obviously, one may *statically* allocate the processing resources to the various tasks including the IBC considering the worst-case processing loads. In this worst-case load, all tasks are

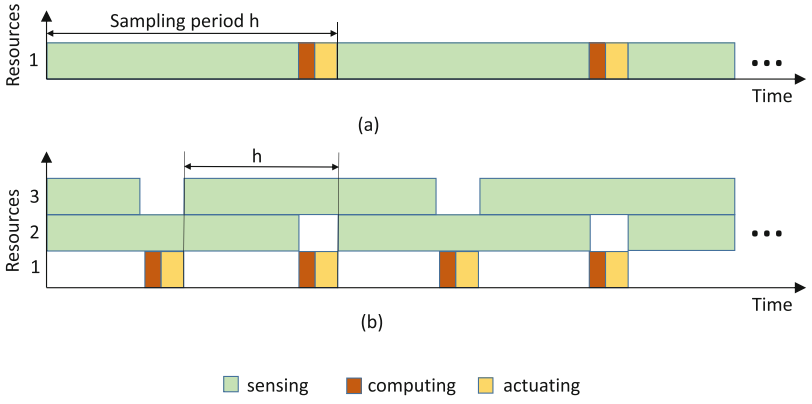


Fig. 9.27 Implementation of data-intensive control systems. (**a**) Sequential implementation of an IBC. (**b**) Pipelined implementation of an IBC

executing together and the IBC demands for a high QoC. In reality, the worst-case processing loads do not always occur and such static allocation leads to an under-utilization of resources. The idea is to *dynamically* allocate resources to the tasks based on both availability of resources and system state (e.g., demand for QoC, number of active tasks). The combination of system state and resource allocation determines the *system scenarios* in this context. At design-time, the individual scenarios are optimized. For example, depending on the number of active tasks, processing resources are allocated to an IBC. The pipelined IBC is modeled, an optimized controller is designed for the pipelined system and implemented for a given allocation. The system may at run-time switch from one scenario to another to achieve system-level optimization. The dynamics resulting from scenario-based optimization translates to switched (linear) control.

xCPS Case Study 13 Consider again the xCPS setup presented in Sect. 2. We focus on the design of an IBC system for controlling Belt 2 speeds (see Fig. 9.3). Recall from Sect. 4.4 that we assumed Belt 2 is the bottleneck component in the system. Assume that four software tasks are implemented on a shared embedded multi-core platform: a supervisory control task (SCT), a Turner task (TT), an indexing table task (ITT), and an image-based control (IBC) divided in sensing, control computation, and actuation. Whenever a piece arrives at Belt 2, SCT is activated to schedule the appropriate actions in the FMS (as explained in Sect. 5.3). This includes generating the reference belt speed signal for the IBC. The IBC system implements the transport actions t1–t4. TT and ITT implement the control of the turner and IT2. SCT also distributes the available processing resources of the embedded platform among the four software tasks. Since the SCT is triggered by the arrival of a new piece, its activation time cannot be predicted. SCT is therefore dedicated to a single core. The activation time of ITT and TT is predictable because the time elapsed between a piece entering the system and reaching the Turner or indexing table IT2 is regulated by the IBC system. TT and ITT can be dynamically assigned to execute on any of the available cores of the platform. Note that they may need to execute concurrently. IBC regulates the speed of Belt 2 using a DC motor. A camera and an image processing algorithm are used as a sensor to measure the speed of the pieces on the belt. The speed of the pieces is kept low when a piece is going through the Turner and Switch 2 actuators and high otherwise. The total processing latency of the image processing algorithm is 400 ms, which potentially limits the controller performance.

For the implementation of the four tasks, we assume a multi-core platform with six cores. One core is dedicated for the control computation and actuation, to keep the control data locally and guarantee the periodicity of the controller; another core is dedicated to the SCT, such that it is activated whenever a new piece arrives; the remaining four cores can be used by TT, ITT, and the image processing needed for IBC. We have three categories of tasks in the system: reconfigurable, non-reconfigurable, and sensing tasks. Non-reconfigurable tasks are statically allocated to a given core, e.g., the SCT and the control computation and actuation tasks. Reconfigurable tasks can be dynamically allocated to different cores, e.g., TT and ITT. Sensing tasks correspond to the image processing algorithm of IBC.

The throughput of the machine is directly affected by the settling time (S) of the IBC system: a shorter S means that the pieces reach each stage of the assembly process faster, increasing the number of assembled pieces per time unit, or the throughput. Therefore, QoC is quantified using S, i.e., $QoC = S^{-1}$. ■

An IBC system implemented on an N-core multi-core platform can be modeled as follows. The platform runs M reconfigurable tasks, P non-reconfigurable tasks, and an IBC task (as explained in the example). The IBC task itself is composed of three tasks: T_s, sensing, T_c, control computation, and T_a, actuation. The dynamic system is modeled as a linear time-invariant system

$$\dot{x}(t) = Gx(t) + Hu(t - \tau),\ y(t) = Cx(t) \tag{9.1}$$

where G, H, and C are state, input, and output matrices, $u(t - \tau)$ is the control signal, $x(t)$ is the system state vector, and $y(t)$ is the output vector. τ is the sensor-to-actuator delay given by

$$\tau = \tau_s + \tau_c + \tau_a \tag{9.2}$$

where τ_s, τ_c, τ_a are the execution times for the sensing, computing, and actuating tasks (taking into account overhead such as communication delays). For an IBC, $\tau_s \gg \tau_c + \tau_a$ due to the high processing latency of the images.

The allocation of processing cores to the sensing tasks or the reconfigurable tasks depends on system requirements. The sampling period h depends on the number of processing cores allocated to the sensing task. Subsequently, the system model has to be derived and the controller needs to be designed. Altogether, they define the system scenarios.

6.2 Scenario Identification and Exploitation

Consider the model introduced in the previous subsection. Since P non-reconfigurable tasks are running on the processing platform, only $N - P$ cores are available to the M reconfigurable tasks and the sensing task. When none of the reconfigurable tasks is active, the supervisory task SCT may allocate all the $N - P$ cores to the T_s task (that may then run $N - P$ pipelined instances in parallel). When one or more reconfigurable tasks are activated, SCT allocates a reduced number of cores to the T_s task. In terms of the system scenario methodology of Chap. 2, all run-time situations with a fixed, given number of cores allocated to the sensing task are clustered in a scenario. We refer to the scenarios as the *scenario with maximal configuration* s_{mc}, with $N_{mc} = N - P$ cores allocated to the sensing task, and the *scenarios with reduced configuration* $s_{rc,\alpha}$ that allocate $N_{rc,\alpha} < N_{mc}$ cores to the sensing task, with α an index to identify the reduced configuration. There may be multiple reduced configurations depending on N and M.

For every scenario, a controller needs to be designed (the exploitation step in the system scenario terminology). The sampling period of the IBC depends on the scenario the system is in. For a given sampling period, the discretized system model is derived and the controller design is done. One of the configurations in a pipelined IBC system, preferably the most frequently occurring one, can be optimized for performance; the other configurations need to be designed considering stability under scenario switching. In the xCPS case, the controller is optimized for performance in the maximal configuration; in the reduced configurations, it is designed to ensure stability. For modeling purposes, it suffices to distinguish only the maximal configuration and an arbitrary reduced configuration.

- **Scenario with maximal configuration s_{mc}—model and controller**: Assume N_{mc} cores are available to the sensing task T_s resulting in a sampling period

$$h_{mc} = \frac{\tau}{N_{mc}} \tag{9.3}$$

Discretizing Equation 9.1 with zero-order hold and sampling period h_{mc}, we obtain the system model of the form

$$x(k+1) = \alpha_{mc}x(k) + \beta_{mc}(\tau')u(k - l_{mc}) + \gamma_{mc}(\tau')u(k - l_{mc} + 1) \tag{9.4}$$

where α_{mc}, β_{mc}, γ_{mc} are the discrete-time state and input matrices of the form

$$\begin{aligned} \alpha_{mc} &= e^{Gh_{mc}}, \\ \beta_{mc}(\tau') &= \int_0^{h_{mc}-\tau'} e^{Gs}Hds, \\ \gamma_{mc}(\tau') &= \int_{h_{mc}-\tau'}^{h_{mc}} e^{Gs}Hds \end{aligned} \tag{9.5}$$

$\tau' = \tau - (l_{mc} - 1)h_{mc}$, $l_{mc} = \left\lceil \frac{\tau}{h_{mc}} \right\rceil$, $u(k - l_{mc})$, and $u(k - l_{mc} + 1)$ are the delayed inputs. Defining the augmented system states

$$z(k) = \left[x^T(k)\ u^T(k - l_{mc})\ u^T(k - l_{mc} + 1) \cdots u(k-1) \right]^T \tag{9.6}$$

we obtain the augmented system model in scenario s_{mc} as

$$z(k+1) = \phi_{mc}z(k) + \Gamma_{mc}u(k) \tag{9.7}$$

where ϕ_{mc} and Γ_{mc} are the discrete-time state and input matrices of appropriate dimension [4]. From Eq. 9.6, it can be noticed that the number of states (i.e., the size of $z(k)$) depends on l_{mc} which further depends on the number of cores allocated to the T_s task. The control law is chosen in the following form:

$$u(k) = K_{mc} z(k) + F_{mc} r \tag{9.8}$$

where K_{mc} is the feedback gain, F_{mc} is the static feedforward gain, and r is the constant reference signal. Clearly, the control gains K_{mc} and F_{mc} depend on l_{mc} which further depends on N_{mc}. The combination of N_{mc}, K_{mc}, and F_{mc} defines the scenario s_{mc}.

- **Scenario(s) with reduced configuration $s_{rc,\alpha}$—model and controller**: Assume that the reduced configuration α has $N_{rc,\alpha} < N_{mc}$ cores are available to the sensing task T_s resulting in a sampling period

$$h_{rc,\alpha} = \frac{\tau}{N_{rc,\alpha}} \tag{9.9}$$

Similar to the scenario s_{mc}, we define the augmented system states

$$z(k) = \left[x^T(k)\ u^T(k - l_{rc,\alpha})\ u^T(k - l_{rc,\alpha} + 1) \cdots u(k-1) \right]^T \tag{9.10}$$

where $l_{rc,\alpha} = \left\lceil \frac{\tau}{h_{rc,\alpha}} \right\rceil$. The number of states in Eq. 9.6 is higher than the number of states in Eq. 9.10. Therefore, we define an extended augmented state vector to make the dimension of the systems identical

$$z(k) = \left[x^T(k)\ u^T(k - l_{mc}) \cdots u^T(k - l_{rc,\alpha})\ u^T(k - l_{rc,\alpha} + 1) \cdots u(k-1) \right]^T, \tag{9.11}$$

and we obtain the augmented system model in scenario $s_{rc,\alpha}$ as

$$z(k+1) = \phi_{rc,\alpha} z(k) + \Gamma_{rc,\alpha} u(k) \tag{9.12}$$

where $\phi_{rc,\alpha}$ and $\Gamma_{rc,\alpha}$ are the discrete-time state and input matrices of appropriate dimension [4]. The control law is chosen in the following form:

$$u(k) = K_{rc,\alpha} z(k) + F_{rc,\alpha} r \tag{9.13}$$

where $K_{rc,\alpha}$ is the feedback gain, $F_{rc,\alpha}$ is the static feedforward gain, and r is the constant reference signal. Similar to s_{mc}, the combination of $N_{rc,\alpha}$, $K_{rc,\alpha}$, and $F_{rc,\alpha}$ defines the scenario $s_{rc,\alpha}$.

The basic idea of the scenario-based control strategy is to switch between s_{mc} and $s_{rc,\alpha}$ depending on the presence of activated reconfigurable tasks, TT and ITT in the xCPS case. When no reconfigurable task is active, the system runs in s_{mc} where all the N_{mc} processing cores are used to compute the T_s task in pipeline. Figure 9.28a shows an example s_{mc} with $N_{mc} = 4$ for the xCPS case. Given the processing time of an image $\tau_s = 400\,\text{ms}$, the sampling period is $h_{mc} = \frac{\tau_s}{4} = 100\,\text{ms}$. The controller gains K_{mc} and F_{mc} in this scenario are designed to optimize performance

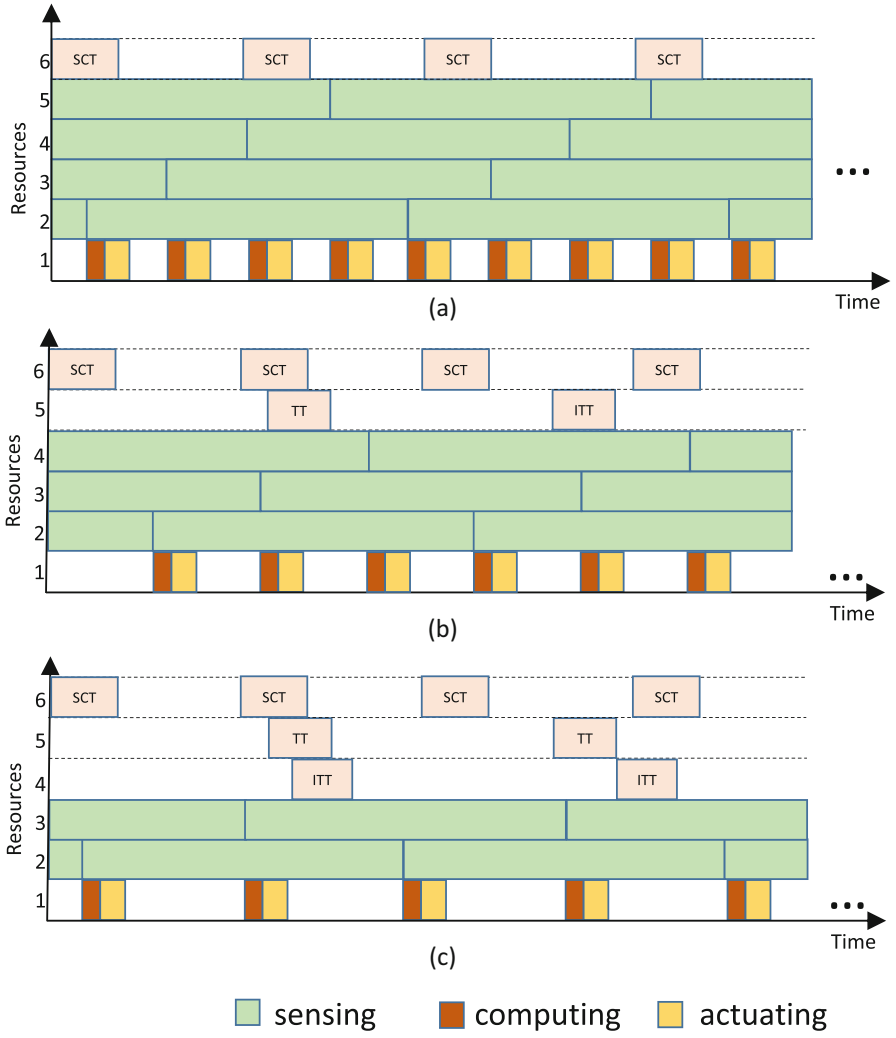


Fig. 9.28 Scenarios in SBPC: s_{mc} and $s_{rc,2}$. (**a**) Scenario s_{mc} with $N_{mc} = 4$. (**b**) Scenario $s_{rc,1}$ with $N_{rc,1} = 3$. (**c**) Scenario $s_{rc,2}$ with $N_{rc,2} = 2$

(i.e., minimize settling time). When ITT or TT is active, the corresponding scenario is denoted as $s_{rc,1}$ with $N_{rc,1} = 3$. In this case, the sampling period is $h_{rc,1} = \frac{\tau_s}{3} = 134$ ms. Figure 9.28b shows an example of $s_{rc,1}$. One core is freed to execute TT or ITT. Likewise, when both ITT and TT are active, scenario $s_{rc,2}$ with $N_{rc,2} = 2$ becomes active. In $s_{rc,2}$, the sampling period is $h_{rc} = \frac{\tau_s}{2} = 200$ ms. Figure 9.28c shows an example of $s_{rc,2}$. Two cores are freed to execute TT and ITT. The controller gains $K_{rc,1}$, $K_{rc,2}$, $F_{rc,1}$, and $F_{rc,2}$ in scenarios $s_{rc,1}$ and $s_{rc,2}$ are designed to ensure stability of the entire switched linear system. That is, the switching between

the scenarios s_{mc}, $s_{rc,1}$, and $s_{rc,2}$ implies switching between systems in Eqs. 9.12 and 9.7 which may potentially destabilize the overall system [37]. To ensure overall stability, $K_{rc,1}$, $K_{rc,2}$ are designed such that a common quadratic Lyapunov function exists between Eqs. 9.12 and 9.7 [24].

6.3 Scenario-Based Pipelined Control (SBPC)

We have seen how to design a scenario-based, pipelined IBC. It is important to note that the activation times of the reconfigurable tasks in the system are assumed to be known in time to allocate the required resources. In the xCPS case, the activation times of TT and ITT are known as soon as an SCT instance is triggered by the arrival of a new piece at Belt 2 (see xCPS case study 13). So the number of active reconfigurable tasks can be used as a parameter to predict the correct scenario at run-time. The switching strategy of our scenario-based pipelined controller (SBPC) is illustrated in Fig. 9.29. In a static design, the system always stays in $s_{rc,2}$ even when there are no other active tasks than the IBC. In the presented scenario-based approach, the system runs under scenario s_{mc} when there are no other tasks, allowing the controller to run with a shorter sampling period h_{mc}. This potentially improves QoC compared to the worst-case given that the sporadic tasks are not always active.

xCPS Case Study 14 We apply SBPC to the xCPS example outlined in xCPS case study 13. We compare our SBPC approach with two other IBC setups: A sequential controller (SC) and a pipelined controller (PC). SC uses a single sensing core as shown in Fig. 9.27a. PC uses a worst-case design with two static sensing cores as shown in Fig. 9.27b. SBPC uses the aforementioned scenarios s_{mc}, $s_{rc,1}$, and $s_{rc,2}$. We consider the case of an inverted top piece entering Belt 2. The belt speed is set initially high (i.e., $8cm/s$) such that the piece arrives quickly at the Turner. Then, the speed has to be set low (i.e., $1cm/s$) such that the Turner can flip the piece. At

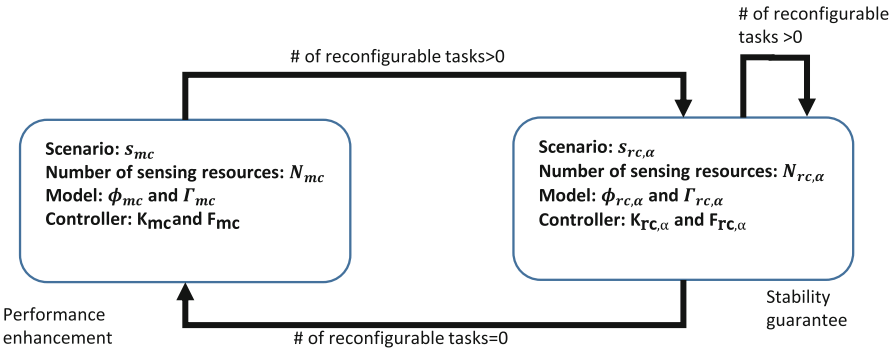


Fig. 9.29 Scenario switching in SBPC

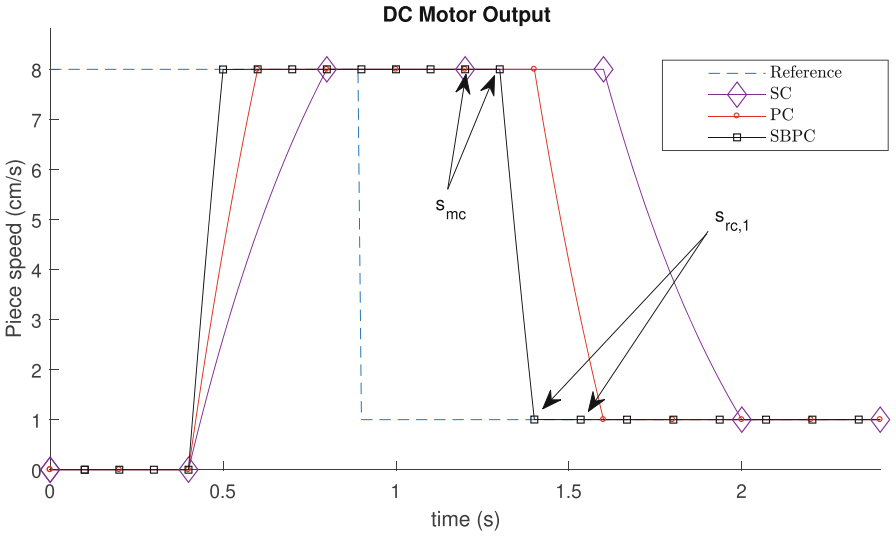


Fig. 9.30 Comparison of IBC strategies

this point, the TT needs to be activated. The IBC then regulates the piece speed on the belt.

The resulting controller performance of the three considered controllers is compared in Fig. 9.30. The simulation shows that the three controllers follow the indicated reference. However, SBPC reaches the reference faster than PC and than SC, which potentially means a higher throughput (i.e., number of assembled pieces per time unit) in xCPS. Note that SBPC runs most of the time in s_{mc}. It only switches to $s_{rc,1}$ when the speed is set to slow, so that the TT can be activated. ■

For FMS like xCPS, ultimately system-level performance metrics like throughput and makespan are most important. It is therefore interesting to analyze the impact of controller performance on system-level performance metrics. Controllers like the ones considered in this section determine action timing in the activity models used for performance analysis. It is therefore sufficient to adapt the timing matrices of activities in order to analyze the impact of controller performance on system-level performance metrics.

xCPS Case Study 15 To evaluate the impact of better QoC on the xCPS throughput, we may use the activity model presented in Sect. 4. To do so, we need to determine the timing information captured in parameters d_2, d_3, and d_4. It has already been explained in xCPS case study 4 how this is done, using Simulink simulations. The timing information is used to compute the system throughput, as presented in xCPS case study 8. This example already showed that SBPC has the best system performance of the considered controllers. A better throughput of 0.111 pieces per second is obtained by a better control performance of SBPC, compared to

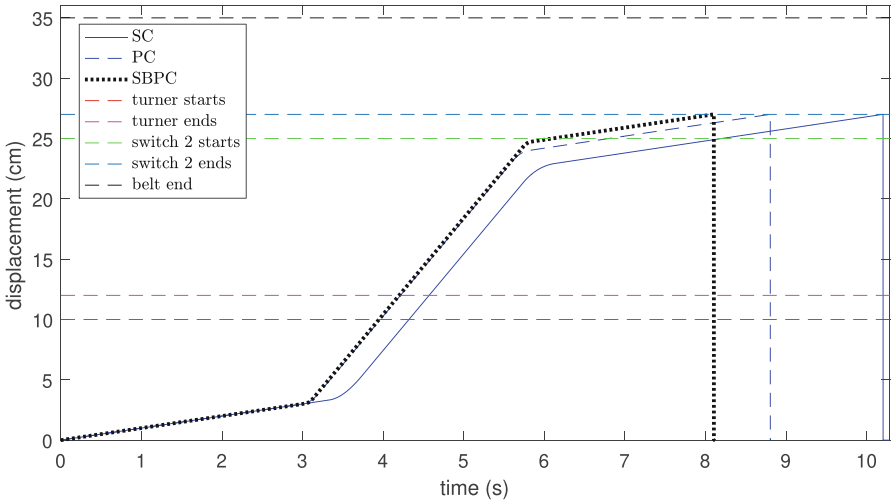


Fig. 9.31 Movement of a bottom piece (B2,TIBTB) on Belt 2

Table 9.1 Example of timing information for (B2,TIBTB)

Timing information	SC	PC	SBPC
d_2	4.55	4.21	4.19
d_3	5.62	4.57	3.89
d_4	0	0	0
Total	10.17	8.78	8.08

All the provided numbers are in seconds

0.103 pieces per second for PC. The sequential setup SC only achieves a throughput of 0.091 pieces per second.

Comparing d_2, d_3, and d_4 in all the IBC setups provides insight in the performance improvement. An example of the timing information of a bottom piece with context (B2,TIBTB) (using the notations of Sect. 5) is shown in Fig. 9.31. This graph is a zoomed in version of graphs as shown in Fig. 9.7. To ease the comparison, the starting point of the piece is set to zero in all the IBC setups. The timing information is summarized in Table 9.1. Note that d_4 is zero because the bottom piece leaves the belt at Switch 2. Clearly, the pieces travel faster when a controller with better QoC is used. SC has the lowest performance among the IBC, with a total transport time of $10.17s$. When comparing PC and SBPC, note that the transport time of a piece is almost the same up to the activation of Switch 2. However, at the beginning of Switch 2, SBPC changes to low speed later than PC, which reduces the transport time of the piece by 0.70. Such a late speed change is possible because of the higher sampling rate and the more aggressive response of SBPC compared to PC. ■

7 Conclusions

This chapter presents three system scenario use cases in FMS design. All the techniques are illustrated on the xCPS assembly line emulator.

Functionally deterministic FMS activities with tightly bounded timing of low-level actions can be represented in activity models of which timing is captured in matrices. This enables fast and accurate system-level performance analysis for metrics such as throughput (productivity) and makespan. System-level performance analysis for the xCPS case study is possible in the order of milliseconds once the activity models are available. This is much faster than the traditionally used fine-grained discrete-event simulations. The approach moreover provides guaranteed bounds on system performance and supports the identification of performance bottlenecks.

The activity models can also be used for scenario-based supervisory controller synthesis and performance optimization of such supervisory controllers. Supervisory controllers ensure safe behavior of FMS, avoiding, e.g., collisions between moving parts or deadlocks due to filled up buffers. Optimizing such supervisory controllers for performance enables system-level performance optimization. Scenario-based, modular modeling in combination with state-space reduction techniques and performance optimization improves scalability of the synthesis, enabling the use of supervisory controller synthesis for a much wider class of systems than traditional synthesis.

Physical actions in FMS are typically implemented using various types of feedforward and feedback control. As a third illustration of the use of scenarios in FMS design, we presented a scenario-based design approach for data-intensive control loops. The readily available multi-processor and multi-core technology enables pipelined design of data-intensive controllers. The scenario-based approach optimizes quality of control when resources are shared with other sporadic or reconfigurable processing tasks. As opposed to worst-case design, the scenarios capture frequent system behaviors and design is optimized for the individual scenarios. By appropriate switching between the scenarios, quality of control is improved. When applied to performance bottlenecks, overall system productivity of the FMS can be improved. The scenario-based pipelined controller design is independent of the scenario-based activity modeling and performance analysis. The latter can be used though to analyze the productivity improvement for the FMS due to an improved quality of control of the pipelined controller compared to a traditional sequential controller.

Acknowledgements This research is supported in part by the Netherlands Organisation for Scientific Research (NWO), through the Robust Cyber-Physical Systems (RCPS) program, projects 12694 and 12697.

References

1. S. Adyanthaya, H. Alizadeh Ara, J. Bastos, A. Behrouzian, R. Medina Sánchez, J. van Pinxten, B. van der Sanden, U. Waqas, T. Basten, H. Corporaal, R. Frijns, M. Geilen, D. Goswami, M. Hendriks, S. Stuijk, M. Reniers, J. Voeten, xCPS: a tool to eXplore cyber physical systems. ACM SIGBED Rev. **14**(1), 81–95 (2016). https://doi.org/10.1145/3036686.3036696
2. R. Alur, *Principles of Cyber-Physical Systems* (MIT Press, Cambridge, 2015)
3. R. Alur, D. Dill, A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
4. K.J. Åström, B. Wittenmark, *Computer-controlled Systems*, 3rd edn. (Prentice-Hall, Bergen, 1997)
5. F. Baccelli, G. Cohen, G. Olsder, J. Quadrat, *Synchronization and Linearity* (Wiley, Hoboken, 1992)
6. J. Baeten, D. van Beek, P. Cuijpers, M. Reniers, J. Rooda, R. Schiffelers, R. Theunissen, Model-based engineering of embedded systems using the hybrid process algebra chi. Electron. Notes Theor. Comput. Sci. **209**, 21–53 (2008)
7. J. Bastos, S. Stuijk, J. Voeten, R. Schiffelers, J. Jacobs, H. Corporaal, Modeling resource sharing using FSM-SADF, in 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015 (IEEE CS, Piscataway, 2015), pp. 96–101
8. J. Bastos, S. Stuijk, J. Voeten, R. Schiffelers, H. Corporaal, Exploiting specification modularity to prune the optimization-space of manufacturing systems, in *Proceedings of the Software and Compilers for Embedded Systems, SCOPES 2018* (ACM, New York, 2018)
9. G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, F. Vaandrager, Minimum-cost reachability for priced time automata, in *Proceedings of the Hybrid systems: computation and control, HSCC 2001*. LNCS, vol. 2034 (Springer, Berlin, 2001), pp. 147–161
10. C.G. Cassandras, S. Lafortune, *Introduction to Discrete Event Systems*, 2nd edn. (Springer, Berlin, 2010)
11. L. Denissen, Image-based Control and Throughput Analysis for Flexible Manufacturing Systems. Master's thesis, Eindhoven University of Technology (2016)
12. E.W. Dijkstra, A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
13. S. Gaubert, Performance evaluation of (max,+) automata. IEEE Trans. Autom. Control **40**(12), 2014–2015 (1995)
14. M. Geilen, Synchronous dataflow scenarios. ACM Trans. Embed. Comput. Syst. **10**(2), 16:1–16:31 (2010)
15. M. Geilen, S. Stuijk, Worst-case performance analysis of synchronous dataflow scenarios, in *Proceedings of the Hardware/Software Codesign and System Synthesis, International Conference, CODES+ISSS 2010* (ACM, New York, 2010), pp. 125–134
16. M. Geilen, J. Falk, C. Haubelt, T. Basten, B. Theelen, S. Stuijk, Performance analysis of weakly-consistent scenario-aware dataflow graphs. J. Signal Process. Syst. **87**(1), 157–175 (2017)
17. M. Hendriks, B. van den Nieuwelaar, F. Vaandrager, Model checker aided design of a controller for a wafer scanner. Int. J. Softw. Tools Technol. Trans. **8**(6), 633–647 (2006)
18. M. Hendriks, J. Verriet, T. Basten, M. Brassé, R. Dankers, R. Laan, A. Lint, H. Moneva, L. Somers, M. Willekens, Performance engineering for industrial embedded data-processing systems, in *Proceedings of the Product-Focused Software Process Improvement, PROFES 2015, International Conference*. LNCS, vol. 9459 (Springer, Berlin, 2015), pp. 399–414
19. J. Huang, J. Voeten, H. Corporaal, Predictable real-time software synthesis. Real-Time Syst. J. **36**(3), 159–198 (2007)
20. J. Huang, J. Voeten, M. Groothuis, J. Broenink, H. Corporaal, A model-driven design approach for mechatronic systems, in *Proceedings of the Application of Concurrency to System Design, ACSD 2007, 7th International Conference* (IEEE CS, Piscataway, 2007), pp. 127–136
21. J.E. Kelley Jr, M.R. Walker, Critical-path planning and scheduling, in *Papers Presented at the December 1–3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference, IRE-AIEE-*

ACM '59 (Eastern) (ACM, New York, 1959), pp. 160–173. https://doi.org/10.1145/1460299.1460318
22. E. Lee, M. Niknami, T. Nouidui, M. Wetter, Modeling and Simulating cyber-physical systems using CyPhySim, in *Proceedings of the International Conference on Embedded Software, EMSOFT 2015* (IEEE CS, Piscataway, 2015)
23. J. Markovski, D. van Beek, R. Theunissen, K. Jacobs, J. Rooda, A state-based framework for supervisory control synthesis and verification, in *Proceedings of the Decision and Control, 2010 49th IEEE Conference on, CDC* (2010), pp. 3481–3486. https://doi.org/10.1109/CDC.2010.5717095
24. O. Mason, R.N. Shorten, On common quadratic Lyapunov functions for stable discrete time LTI systems. IMA J. Appl. Math. **69**, 271–283 (2002)
25. MathWorks, *Simulink: Simulation and Model-Based Design* (2018). https://nl.mathworks.com/products/simulink.html
26. R. Medina, S. Stuijk, D. Goswami, T. Basten, Reconfigurable pipelined sensing for image-based control, in *Industrial Embedded Systems, 11th IEEE International Symposium, SIES 2016* (IEEE CS, Piscataway, 2016), pp. 1–8
27. Y. Narahari, N. Viswanadham, A Petri net approach to the modelling and analysis of flexible manufacturing systems. Ann. Oper. Res. **3**, 449–472 (1985)
28. Océ Technologies, *Océ VarioPrint i300* (2018). https://www.canon-europe.com/business-printers-and-faxes/cut-sheet-colour-printers/varioprint-i300/
29. L. Ouedraogo, R. Kumar, R. Malik, K. Akesson, Nonblocking and safe control of discrete-event systems modeled as extended finite automata. IEEE Trans. Autom. Sci. Eng. **8**(3), 560–569 (2011). https://doi.org/10.1109/TASE.2011.2124457
30. D. Peled, P. Pelliccione, P. Spoletini, Model checking, in *Wiley Encyclopedia of Computer Science and Engineering* (Wiley, Hoboken, 2009)
31. A. Rahatulain, T. Qureshi, M. Onori, Modeling and simulation of evolvable production systems using Simulink/SimEvents, in *Proceeding of the 40th Annual Conference of the IEEE Industrial Electronics Society, IECON 2014* (IEEE CS, Piscataway, 2014), pp. 2591–2596
32. P.J.G. Ramadge, W.M. Wonham, Supervisory control of a class of discrete event processes. SIAM J. Control. Optim. **25**(1), 206–230 (1987)
33. P.J.G. Ramadge, W.M. Wonham, The control of discrete event systems. Proc. IEEE **77**(1), 81–98 (1989)
34. M. Silva, R. Valette, Petri nets and flexible manufacturing, in *Advances in Petri Nets 1989, LNCS 424* (Springer, Berlin, 1989), pp. 374–417
35. M. Skoldstam, K. Åkesson, M. Fabian, Modeling of discrete event systems using finite automata with variables, in *Proceeding of the Decision and Control, 2007 46th IEEE Conference on, CDC* (2007), pp. 3387–3392. https://doi.org/10.1109/CDC.2007.4434894
36. S. Stuijk, M. Geilen, B. Theelen, T. Basten, Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications, in *Proceeding of the Embedded Computer Systems: Architectures, Modeling, and Simulation, International Conference, IC-SAMOS 11* (IEEE CS, Piscataway, 2011), pp. 404–411
37. Z. Sun, S.S. Ge, *Stability Theory of Switched Dynamical Systems* (Springer, Berlin, 2011)
38. Uppaal (2018). http://www.uppaal.com
39. J. Valencia, D. Goswami, K. Goossens, Composable platform-aware embedded control systems on a multi-core architecture, in *Proceeding of the Digital System Design, 2015 Euromicro Conference on* (IEEE, Piscataway, 2015), pp. 502–509
40. B. van der Sanden, Performance Analysis and Optimization of Supervisory Controllers. Ph.D. thesis (Eindhoven University of Technology, Eindhoven, 2018)
41. B. van der Sanden, M. Reniers, M. Geilen, T. Basten, J. Jacobs, J. Voeten, R. Schiffelers, Modular model-based supervisory controller design for wafer logistics in lithography machines, in *Proceeding of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, MODELS 2015* (IEEE CS, Piscataway, 2015), pp. 416–425
42. B. van der Sanden, J. Bastos, J. Voeten, M. Geilen, M. Reniers, T. Basten, J. Jacobs, R. Schiffelers, Compositional specification of functionality and timing of manufacturing systems,

in *Forum on Specification & Design Languages, FDL 2016, Proceeding IEEE* (CS, Piscataway, 2016). https://doi.org/10.1109/FDL.2016.7880372
43. B. van der Sanden, M. Geilen, M. Reniers, T. Basten, Partial-order reduction for performance analysis of max-plus timed systems, in *Proceeding of the Application of Concurrency to System Design, 18th International Conference, ACSD 2018* (IEEE CS, Piscataway, 2018), pp. 40–49
44. J. Xing, B. Theelen, R. Langerak, J. van de Pol, J. Tretmans, J. Voeten, UPPAAL in practice: quantitative verification of a RapidIO network, in *Proceeding of the Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2010, 4th International Symposium*. LNCS, vol. 6416 (Springer, Berlin, 2010), pp. 160–174.
45. M. Zhou, K. Venkatesh, *Modeling, Simulation, and Control of Flexible Manufacturing Systems: A Petri Net Approach* (World Scientific, Singapore, 1999)

Index

F. Catthoor et al. (eds.), *System-Scenario-based Design Principles and Applications*, https://doi.org/10.1007/978-3-030-20343-6