DISS. ETH NO. 22211

Programming Framework for Reliable and Efficient Embedded Many-Core Systems

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

LARS URS SCHOR

MSc ETH EEIT, ETH Zurich

born on 19.10.1984
citizen of Subingen SO

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Soonhoi Ha, co-examiner

2014

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 148

Lars Urs Schor

Programming Framework for Reliable and Efficient Embedded Many-Core Systems

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

Diss. ETH No. 22211

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Soonhoi Ha, co-examiner

Examination date: 10.10.2014

Abstract

Many-core Systems-on-Chip (SoCs) are of increasing significance in the domain of high-performance embedded computing systems where high performance requirements meet stringent timing constraints. The high computing power offered by many-core SoCs, however, does not necessarily translate into high performance. On the one hand, the use of deep submicrometer process technology to fabricate SoCs imposes a major rise in the power consumption per unit area so that many-core SoCs face various thermal issues. On the other hand, many-core SoCs are often not capable of fully exploiting the provided hardware parallelism due to runtime variations of executing applications.

In this thesis, we focus on the system-level design of streaming-oriented embedded systems. We tackle the above described challenges by proposing a model-driven development approach for many-core SoCs. The developed high-level programming model specifies an application as a network of autonomous processes that can only communicate over point-to-point First-In First-Out (FIFO) channels. We show that the properties of the proposed programming model can be leveraged to develop a design, optimization, and synthesis process for embedded many-core SoCs that enables the system to utilize its computing power efficiently.

Specifically, the following contributions are presented in this thesis:

- A scenario-based design flow for mapping a set of dynamically interacting streaming applications onto a heterogeneous many-core SoC is described.

- A novel semantics for specifying streaming applications is introduced that abstracts several possible application granularities in a single high-level specification.

- A systematic approach to exploit the multi-level parallelism of heterogeneous many-core architectures is proposed and applied to develop a general code synthesis framework to execute streaming applications on heterogeneous systems.

- A high-level optimization framework for mapping streaming applications onto embedded many-core architectures and optimizing a system design with respect to both performance and worst-case chip temperature is proposed.

Zusammenfassung

Vielkernprozessorsysteme gewinnen im Bereich der eingebetteten Hochleistungs-Rechensysteme vermehrt an Bedeutung, da diese Systeme gleichzeitig hohe Rechenanforderungen und strikte Echtzeitbedingungen einhalten müssen. Vielkernprozessorsysteme können jedoch die vorhandene Rechenleistung oftmals nicht ausnutzen. Einerseits treten aufgrund der hohen Leistungsdichte verschiedene Temperatur bezogene Probleme auf. Anderseits ist das System oftmals nicht in der Lage die vorhandene Hardware-Parallelität auszunutzen, da das System beispielsweise nicht auf Veränderungen zu reagieren vermag, welche während der Laufzeit auftreten.

Die vorliegende Dissertation befasst sich mit der Entwicklung von eingebetteten Systemen für Streaming Anwendungen. Die oben genannten Schwierigkeiten werden mit Hilfe eines neu entwickelten modellgetriebenen Entwicklungsansatzes für Vielkernprozessorsysteme behandelt. Das dabei benutzte Programmiermodel spezifiziert jede Applikation als ein Netzwerk von unabhängigen Prozessen, welche ausschliesslich über gerichtete Kanäle kommunizieren können. Es wird gezeigt, wie man die fundamentalen Eigenschaften des vorgeschlagenen Programmiermodelles ausnutzen kann, um einen Entwurfs-, Optimierungs- und Code-Generierungsprozess zu entwickeln, welcher die Rechenleistung von Vielkernprozessorsystemen effektiv und effizient ausnützt.

Dabei werden die folgenden Resultate in dieser Dissertation präsentiert:

- Ein auf Systemzustände basierender Entwurfsablauf für die Ausführung von mehreren dynamisch interagierenden Streaming Applikationen auf Vielkernprozessorsystemen wird präsentiert.

- Eine neue Semantik für die Spezifikationen von Streaming Applikationen wird eingeführt. Die neue Semantik abstrahiert verschiedene Granularitäten in einer einzelnen Applikationsspezifikation.

- Ein systematischer Ansatz, um die mehrstufige Hardware-Parallelität von heterogenen Vielkernarchitekturen auszunützen, wird vorgeschlagen. Der Ansatz wird verwendet, um eine Laufzeitumgebung zu entwickeln, welche Streaming Applikationen auf heterogenen Systemen ausführen kann.

- Eine Berechnungsmethodik für die Zuweisung von Streaming Applikationen auf eingebettete Vielkernarchitekturen wird vorgeschlagen, welche das System sowohl hinsichtlich seiner Performanz als auch seiner maximalen Temperatur optimiert.

Acknowledgement

This thesis would not have been possible without the help and support of many people who supported me during my studies in various ways. First of all, I would like to express my gratitude to Prof. Dr. Lothar Thiele for his guidance, support, and inspiring advice. I would also like to thank my co-examiner Prof. Dr. Soonhoi Ha for his willingness to review this thesis and to serve on my committee board.

Many other people directly or indirectly contributed to the content of this dissertation. I would like to thank my collaborators (and supervisors of my master thesis) Dr. Iuliana Bacivarov and Prof. Dr. Hoeseok Yang with whom I worked very closely and who significantly contributed to the work presented in this thesis. Furthermore, I would like to thank my officemate Devendra Rai for the successful teamwork throughout the years. Central for this thesis were also the collaboration with Shin-haeng Kang, Tobias Scherer, and Andreas Tretter for which I am very grateful.

Furthermore, many thanks go to all current and former colleagues of the Computer Engineering group at ETH Zurich, especially Ben Buchli, Georgia Giannopoulou, Andres Gomez, Pengcheng Huang, Dr. Pratyush Kumar, and Dr. Nikolay Stoimenov for the collaboration in teaching and research throughout the years. I am also beholden to Dr. Wolfgang Haid and Dr. Kai Huang who encouraged me to become a PhD student at the Computer Engineering group. I also had the pleasure to collaborate with many people from the EURETILE collaboration and I would like to thank all of them for this experience.

A number of internships, semester theses, and master theses have been conducted in the context of this dissertation. I would like to thank: Matthias Baer, Jan Bernegger, Gino Brunner, Stefan Draskovic, Pierre Ferry, Etienne Geiser, James Guthrie, Erwin Herrsche, Georgios Kathareios, Alamanda Poorna Chandra Sekhar, and Felix Wermelinger.

Last but not least, I would like to express my gratitude to my family and friends for supporting me during the last four years. I am indebted and grateful to my parents Pia and Urs, and my brother Ivo for their constant support and encouragement that I received during my years at ETH. Most importantly, the biggest thank goes to my girlfriend Jeanine for her love, for encouraging me over and over again, and for her tireless support since more than a decade.

The work presented in this thesis was supported in part by the EU FP7 project EURETILE (grant number 247846), the Korean-Swiss science and technology cooperation program, and an Intel PhD Fellowship. This support is gratefully acknowledged.

Contents

Abstract i

Zusammenfassung iii

Acknowledgement v

1 Introduction 1
- 1.1 Trends in Embedded Systems 2
- 1.2 Model-Driven Development 5
- 1.3 Challenges in the Design of Many-Core Systems-on-Chip 8
- 1.4 Aim of this Thesis 11
- 1.5 Thesis Outline and Contributions 11

2 Scenario-Based System Design 17
- 2.1 Introduction 17
- 2.2 Related Work 20
- 2.3 Architecture Model 21
- 2.4 Scenario-Based Model of Computation 24
- 2.5 Hybrid Mapping Optimization 30
- 2.6 Exemplified Implementation Targeting Intel SCC 38
- 2.7 Evaluation 43
- 2.8 Summary 53

3 Expandable Process Networks 55
- 3.1 Introduction 55
- 3.2 Related Work 57
- 3.3 System Design 60
- 3.4 The Semantics of Expandable Process Networks 62
- 3.5 Application and Mapping Optimization 70
- 3.6 Application Transformation 75
- 3.7 Evaluation 86
- 3.8 Summary 97

4 Exploiting Multi-Level Hardware Parallelism 99
4.1 Introduction . . . 99
4.2 Related Work . . . 101
4.3 OpenCL Background . . . 103
4.4 Problem and Approach . . . 104
4.5 System Specification . . . 108
4.6 OpenCL Synthesizer . . . 111
4.7 Runtime-System . . . 113
4.8 Evaluation . . . 117
4.9 Summary . . . 124

5 Thermal-Aware System Design 129
5.1 Introduction . . . 129
5.2 Related Work . . . 133
5.3 Mapping Optimization Framework . . . 134
5.4 System Model . . . 138
5.5 Peak Temperature Analysis . . . 145
5.6 Generation and Calibration of Analysis Models . . . 160
5.7 Minimizing the Peak Temperature . . . 165
5.8 Evaluation . . . 168
5.9 Summary . . . 182

6 Conclusion 185
6.1 Main Results . . . 185
6.2 Possible Future Directions . . . 187

List of Acronyms 189

Bibliography 191

List of Publications 215

Curriculum Vitae 219

1

Introduction

This thesis presents novel techniques for the design and analysis of future many-core Systems-on-Chip (SoCs). Many-core SoCs are massively parallel systems that integrate hundreds to thousands of cores of different types on a single chip [Bor07]. Many-core SoCs are of increasing significance in the domain of high-performance embedded computing systems [Wol14] where high performance requirements meet stringent timing constraints. An example of this trend is the automotive industry, where computing systems embedded into autonomous cars must solve multi-sensorial data fusion and artificial intelligent problems in real-time [KRJ13].

However, the available computing power does not necessarily translate into high performance as many-core SoCs face thermal issues or do not fully exploit the provided hardware parallelism. As a result, new design and analysis techniques must be developed that tackle the challenges arising when designing many-core SoCs. First, one of the most significant barriers towards high performance systems is the thermal wall [HFFA11]. The increased performance, coupled with a technology reduction, imposes a major rise in the power consumption per unit area so that the reliability of the system is threatened by various thermal issues including high chip temperatures. Second, the offered computing power can only be exploited if the system reacts to runtime variations by dynamically reconfiguring the applications and if the system leverages the peculiarities of the individual processing cores.

This thesis proposes a systematic approach to design reliable and efficient many-core SoCs. The contributions include a design flow for mapping dynamically interacting streaming applications onto many-core

SoCs, techniques to efficiently exploit the multi-level hardware parallelism of heterogeneous many-core SoCs, and thermal-aware optimization strategies.

We continue this chapter with a review of the current trends in embedded software and hardware. In Section 1.2, we discuss the benefits of using a model-driven development approach to design embedded systems. Afterwards, in Section 1.3, we list the challenges that have to be mastered in the design of reliable and efficient many-core SoCs. In Section 1.4, we state the aim of this thesis. Finally, in Section 1.5, we give the outline and summarize the contributions of the thesis.

1.1 Trends in Embedded Systems

In order to meet the performance requirements of the next generation embedded applications, embedded hardware is currently undergoing a major change from uni-core processors to many-core SoCs. In this section, we review this evolution by discussing the current trends in embedded software and hardware separately.

1.1.1 Embedded Software

In recent years, embedded systems entered new fields of applications that are often centered around the processing of data streams [TKA02, GBC06, KBC09, Thi09, Hai10, MC14]. Examples include real-time speech recognition, embedded computer vision, and all kinds of advanced features that are nowadays ubiquitous in cars as, for instance, the anti-lock braking system, electronic stability control, or adaptive cruise control [CLB+12]. Moreover, with autonomous cars on the horizon, embedded systems must soon be able to solve a multitude of computer vision and artificial intelligent problems in real-time [KRJ13].

Usually, such streaming applications are repeatedly reading, processing, and writing out individual stream objects. A key characteristic is that the data processing can be split up into almost independent processing stages that operate on single data items such as bits, audio frames, or video frames. Due to this characteristic, streaming applications are well-suited for execution on massively parallel platforms. Even though some of the concepts proposed in this thesis might be applied to other classes of applications, we are targeting the domain of streaming applications in this thesis.

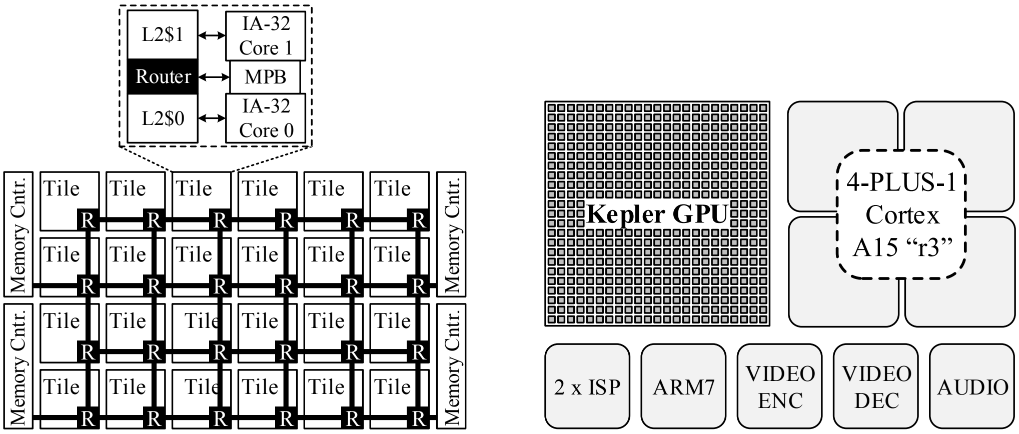
Besides entering new fields of applications, embedded systems are also becoming multi-functional, e.g., by featuring multiple applications that share the system. Smartphones, for instance, are not only used to make phone calls anymore, but provide almost the same possibilities as general-purpose computing systems. As the functionality of the system can change over time, embedded systems must be able to deal with complex dynamic interactions between the applications. Moreover, with each application having its own real-time constraints and quality of service requirements, the system designer has to make sure that each application meets its individual requirements independently of other applications.

1.1.2 Embedded Hardware

The computational demand of modern embedded applications cannot be fulfilled anymore by uni-core processors without expensive cooling systems. As a result, embedded system designers started to use Multiprocessor Systems-on-Chip (MPSoCs) that addressed this challenge partially by spreading the workload over multiple processors. By increasing the number of processors or processing cores rather than the clock frequency, the performance gain was translated into a slower growth in power consumption.

Nonetheless, MPSoCs are only an interim solution for embedded systems as the computational demand of future embedded applications is far beyond the potential of state-of-the-art embedded MPSoCs [WJM08]. A promising solution is the use of many-core SoCs where the workload is spread over hundreds of cores [MGC08]. First prototypes of such platforms have been announced in 2010. The Intel Single-chip Cloud Computer (SCC) processor [HDH+10], for instance, consists of 24 tiles, each with two cores, that are organized into a 4 × 6 grid and linked by a 2D mesh on-chip network. Figure 1.1a shows the schematic outline of the processor. Another example of such a processor is the TILERA TILE-Gx100 [Ram11] that integrates 100 general-purpose processing cores into a single chip.

While the first generation of these many-core systems was not suited for mobile embedded applications due to power consumption in the order of a few tens of watts, the next generation is already much more power efficient. The STMicroelectronics P2012 platform [BFFM12, MBF+12] is a massively parallel processor that can be powered by a battery. Its first silicon implementation integrates 64 cores that are organized in four clusters and each cluster has a worst-case power consumption of about 400 mW. While the first silicon implementation of the P2012 platform just integrated general-purpose processing cores, a recent extension, the He-P2012 platform [CPMB14], augments the clusters with a set of hardware accelerators that communicate with the general-purpose cores over a shared scratchpad. Toshiba recently announced a low-power many-core SoC with two 32-core clusters [XTU+12, MXK+13] that are connected



(a) Intel SCC processor. **(b)** Nvidia Tegra K1 mobile processor.

Fig. 1.1: Schematic outline of the Intel SCC processor [HDH+10] and the Nvidia Tegra K1 mobile processor [NVI14]. The Intel SCC processor is used as target platform in various case studies in this thesis and its architectural design is detailed in Section 2.6.

by a tree-based Network on Chip (NoC) communication architecture. Each core is running at 333 MHz, resulting in a power consumption per cluster of less than 1 W. Finally, the KALRAY MPPA MANYCORE processor [DGL+13] is a programmable many-core processor that is composed of an array of clusters whereby each cluster is composed of 16 processing cores. The first implementation of the KALRAY MPPA MANYCORE processor is composed of 16 clusters thereby integrating 256 cores on a single chip. Current predictions expect the first KALRAY MPPA MANYCORE processor with more than 1000 cores to be released by 2015 [KAL14].

Besides the general trend to move to processors with hundreds of cores, processors in the embedded domain are predicted to become heterogeneous [CFGK13], aiming to achieve significant speedups and higher energy-efficiency. Typically, such heterogeneous many-core SoC platforms consist of a host processor that is coupled with various hardware accelerators [SABC10]. In fact, Graphics Processing Units (GPUs)-like accelerators are increasingly used in mobile embedded systems to offload data-intensive computational kernels. Recent studies have reported that significant speed-ups can be achieved by executing part of the application on the GPU [GCC+14].

Such accelerator-based SoCs are already widely used in mobile devices like smartphones or tablet computers. However, they have recently undergone a steep increase in the number of integrated processing cores [Cas13]. To illustrate the concept of such accelerator-based SoCs, we consider two state-of-the-art mobile processors. The Qualcomm first octa-core SoC, the Snapdragon 615 [Qua14b], integrates an octa-core ARM Cortex A53 Central Processing Unit (CPU) and an Adreno 405 GPU. It is worth to note that four cores of the CPU are clocked at 1.8 GHz and the remaining cores are only clocked at 1.0 GHz. Nvidia's next generation Tegra processor K1 [NVI14], schematically outlined in Fig. 1.1b, integrates a 4-plus-1 quad-core ARM Cortex A15 CPU and a Kepler GPU with 192 cores. Each core of the CPU can be enabled and disabled independently for maximum performance and energy efficiency. However, like any streaming processor, the GPU can only be fully exploited if all cores perform the same operation simultaneously on multiple data sources.

In summary, we can observe that embedded hardware is currently undergoing a major change. Embedded processors will soon integrate hundreds of cores that are split into clusters and connected by a hierarchically organized communication architecture. To meet the stringent energy requirements of mobile devices, the cores will not be of the same type and some processing cores can only be fully exploited if the same instructions are applied concurrently to multiple data sources.

1.2 Model-Driven Development

A recently advocated strategy to design reliable and efficient systems is to restrict the application to a certain (high-level) programming model. This enables the automatic generation and optimization of the platform-dependent implementation by leveraging properties of the programming model. Such a strategy is commonly referred to as model-driven software development [Sel03]. In this section, we review recent efforts towards the model-driven development of embedded systems.

1.2.1 Y-Chart Paradigm

The Y-chart paradigm [KDVvdW97] is a design flow model that implements the ideas of a model-driven software development. It is based on the orthogonalization of concern [KNRSV00], which proposes to master the complexity of embedded systems by separating parts of the design process. The basic idea of the Y-chart paradigm, schematically outlined in Fig. 1.2, is to separate the specification of the application, architecture, and mapping, whereby the mapping specification describes how the application is executed spatially and temporally on the architecture. During the design process, the application, architecture, and mapping specifications are iteratively refined until the quality of service requirements and the real-time constraints of the system are fulfilled.

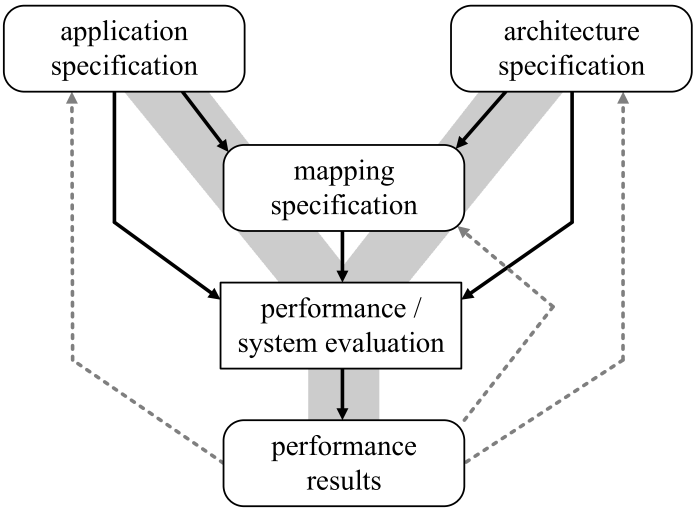

application specification
architecture specification
mapping specification
performance / system evaluation
performance results

Fig. 1.2: Y-chart paradigm [KDVvdW97] to design embedded systems.

Due to the popularity of streaming applications, various models of computation to specify streaming applications have been proposed in the past few years. Most of these models have in common that they split the streaming application into autonomous processes and their interconnections. Kahn Process Networks (KPNs) [Kah74], for instance, specify an application as a set of autonomous processes that communicate through point-to-point First-In First-Out (FIFO) channels. In the context of KPNs, each process basically represents a monotonic mapping of one or more input streams to one or more output streams whereby monotonicity is obtained by having blocking and destructive read access to the channels. The Synchronous Dataflow (SDF) [LM87] model of computation is a restricted version of KPNs. It enables static scheduling analysis at compile time by restricting each process to produce and consume a fixed number of tokens in every iteration. Note that in the context of SDF, processes are commonly referred to as actors.

1.2.2 Design Flows for Mapping Streaming Applications onto Parallel Systems

The KPN and SDF models of computation have been the basis for various high-level embedded system design flows for mapping streaming applications onto parallel systems, see [BHHT10] for an overview. In the following, we review some of them in alphabetic order, with emphasis being given to the programming model and the considered optimization strategies.

CA-MPSoC [SKS$^+$10] is an automated design flow for mapping multiple applications onto communication assist multi-processor systems. Applications are specified as SDF graphs and the dynamic behavior of the system is modeled as a set of use-cases, whereby each use-case represents a combination of concurrently executing applications.

The CompSOC [GAC$^+$13] platform and its associated design flow map multiple concurrent applications onto multi-processor systems by assigning each application its virtual execution platform. Applications are therefore completely isolated, which enables the independent design and verification of each application. The design flow has been automated for real-time Cycle-Static Dataflow (CSDF) applications and the integration of other models of computation has been investigated theoretically.

The DAEDALUS design flow [NSD08] enables the automated mapping of multimedia applications onto embedded MPSoCs and includes design space exploration, system-level synthesis and system prototyping. Its input is a static affine nested loop program, which is automatically converted to a KPN using the KPNgen tool [VNS07]. Multi-application support has been added to DAEDALUS [BZNS12] by using real-time scheduling algorithms to temporally isolate the applications.

The Distributed Operation Layer (DOL) framework [TBHH07] maps a parallel application specified as a KPN onto tiled MPSoCs. During design space exploration, the Modular Performance Analysis (MPA) framework [WTVL06] is used to analyse the performance of the system. The MPA framework uses real-time calculus [TCN00] to reason about timing properties of data flows in queuing networks, thereby providing worst-case timing guarantees for all processes running on the system. The DOL framework is also the starting point of this thesis.

Koski [KKO$^+$06] is a design flow for MPSoCs that uses the Unified Modeling Language (UML) to specify KPNs. For performance analysis, Koski uses simulation that can be calibrated automatically based on the profiling of a small set of calibration implementations. The architecture exploration is based on the composition of subsystems and each subsystem is optimized separately for a particular objective.

The MAPS design flow [CCS$^+$08] maps dataflow applications onto MPSoCs. The design flow's input are a set of sequential programs written in a variant of C called C for Process Networks (CPN). After parallelizing these programs, the MAPS design flow generates performance analysis models based on KPNs and uses a simulation-based composability analysis to provide performance guarantees on the target platform. Multi-application support is provided in MAPS [CVS$^+$10, CLA13] by specifying use-cases, i.e., subsets of running applications, and analyzing all use-cases individually.

The PeaCE framework [HKL+07] is an integrated hardware/software co-design framework for embedded multimedia systems. In order to specify the system behavior, a combination of three models of computation is employed. The Synchronous Piggybacked Dataflow (SPDF) model is used for computation tasks, an extended finite state machine model for control tasks, and a task model to describe the interactions between the tasks. System verification and optimization is conducted by simulating the system on a virtual prototyping environment. Recently, the framework has been extended into an integrated embedded system design environment called HOPES [JLK+14].

SHIM [ET06] is a design flow targeting reconfigurable hardware and multi-processors. Applications are specified in a domain-specific language that is basically a restriction of KPNs. In particular, a rendezvous communication protocol [Hoa85] that allows two communicating processes to advance only when they are synchronized, is elaborated.

The StreamIt design flow [TKA02] is facilitate to map large streaming applications onto various target platforms ranging from MPSoCs to clusters of workstations. It uses the SDF model of computation for application specification so that a wide range of optimization techniques can be applied to the application (e.g., [SGA+13]). Over the past years, a large benchmark suite for StreamIt has been developed that has been used by the embedded system community to characterize the challenges of designing and implementing streaming applications [GTA06, TA10].

Although a huge amount of work has been devoted into developing efficient design flows for multi-core platforms, these techniques will not necessarily be able to exploit the massive parallelism of future many-core SoCs [Vaj11]. Consequently, in this thesis, we try to tackle the challenges that arise when a set of dynamically interacting streaming applications is executed on massively parallel hardware platforms.

1.3 Challenges in the Design of Many-Core Systems-on-Chip

Designing reliable embedded many-core SoCs is difficult. The system designer has not only to cope with new challenges that arise from the massive computational demand of future embedded applications, but also with the hardware parallelism of many-core SoCs. In particular, the following challenges are addressed within this thesis.

1.3.1 Multi-Functional Embedded Systems

Embedded systems are becoming multi-functional. Multiple applications, which appear and disappear dynamically over time, are concurrently executed on the same system. For instance, an intelligent parking assistant application of a semi-autonomous vehicle is executed on the same system as other safety-critical applications are executed and the parking assistant application must only be executed when the vehicle is being parked. To efficiently exploit the offered computing power, the system must react to runtime variations by adapting the mapping of the applications onto the architecture.

As a result, the resources available to a single application can change over time, which in turn complicates the selection of the right degree of application parallelism at the time that the application is specified. On the one hand, programming the application with too many parallel processes might result in inefficient implementations of the application due to overheads in scheduling and inter-process communication. On the other hand, setting the number of parallel processes too small limits the number of cores that the application can use at once. Therefore, the optimal degree of application parallelism for maximum performance depends on the available computing resources and may change over time.

The first challenge tackled in this thesis is the extension of the design process of embedded systems to support multiple applications. Thus, the question is:

> *How to design dynamic multi-functional embedded many-core systems so that reasoning about correctness and performance is enabled at design time and the hardware resources are efficiently exploited?*

To this end, we introduce a design flow for mapping multiple streaming applications onto many-core SoCs in Chapter 2. Afterwards, we extend this formalism and propose in Chapter 3 a streaming programming model that abstracts several possible application granularities in a single specification enabling the selection of the best degree of application parallelism at runtime.

1.3.2 Multi-Level Hardware Parallelism

Future many-core SoCs are predicted to become heterogeneous. Such systems consist of a wide variety of different processing cores including general-purpose processing units, GPUs, or accelerators. Many of these processing cores are able to process multiple threads in parallel and some of them are only able to fully exploit their performance if the same instruction is applied simultaneously to multiple data sources.

Exploiting this multi-level parallelism is challenging, in particular if the application specification does not coincide with the available hardware parallelism. Moreover, the final performance of the system is often reliant on many low-level implementation details, which makes programming heterogeneous systems difficult and error-prone. Inter-process communication, for instance, can drastically reduce the performance if the memory location is not carefully selected. Consequently, the second challenge tackled in this thesis covers the hardware parallelism of heterogeneous many-core SoCs, thereby asking the following question:

> *How to efficiently utilize the multi-level parallelism of heterogeneous many-core SoCs with a high-level programming model?*

Chapter 4 tackles this challenge by describing a design flow that systematically leverages different kinds of application parallelism to exploit the different levels of hardware parallelism.

1.3.3 Temperature-Aware System Design

Nowadays, the thermal wall is recognized as one of the most significant barriers towards high performance systems [HFFA11]. In particular, the demand for increased performance, the technology reduction, and the reduced sizes impose a major rise in the power consumption per unit area so that many-core SoCs face various thermal issues including high chip temperatures.

Reactive thermal management techniques, which are considered as efficient tools for temperature control in general-purpose computing systems, keep the maximum temperature under a given threshold by stalling or slowing down the processor [DM06, KSPJ06]. However, causing a significant performance degradation, reactive thermal management techniques are often undesirable in embedded systems, in particular when real-time constraints are tackled. Therefore, providing guarantees on maximum temperature is as important as functional correctness and timeliness when designing embedded many-core SoCs. A viable approach to provide such guarantees is to adopt system-level mechanisms, and include thermal analysis techniques already at design time. The question hereby is:

> *How to rule out system designs that do not conform to peak temperature requirements already in early design stages of embedded many-core SoCs?*

We will tackle this challenge in Chapter 5 by exploring thermal-aware optimization strategies for embedded many-core SoCs.

1.4 Aim of this Thesis

With this work, we aim to defend the following thesis:

> *The offered computing power of many-core SoCs does not necessarily translate into high performance as the system hits the thermal wall or does not fully exploit the provided hardware parallelism. It is possible to utilize the properties of streaming programming models to improve the design, optimization, and synthesis process of embedded systems so that the computing power of many-core SoCs is efficiently exploited.*

1.5 Thesis Outline and Contributions

This thesis proposes a model-driven development approach to design reliable and efficient many-core SoCs. We target the domain of streaming applications and employ process networks as model of computation. The resulting design flow is illustrated in Fig. 1.3. In this view, the design flow can be considered as a multi-application extension of the Y-chart design paradigm [KDVvdW97] introduced in Section 1.2.

Major contributions of this thesis are to be found in the application specification, the application and mapping optimization, and the performance analysis. It provides, on the one hand, a set of novel techniques for the design of future many-core SoCs tackling the challenges described in Section 1.3. On the other hand, the thesis demonstrates how the proposed techniques can be integrated seamlessly into the overall design flow. Specifically, Chapter 2 introduces the overall design flow for mapping multiple streaming applications onto many-core SoCs. Chapter 3 proposes a technique to find the optimal degree of application parallelism for maximum performance. Chapter 4 demonstrates how the multi-level hardware parallelism of many-core SoCs can systematically be exploited. Finally, Chapter 5 introduces the temperature as an additional evaluation metric in the optimization process of an embedded system. The contributions of this thesis are summarized as follows:

Chapter 2: Scenario-Based System Design

A design flow for mapping multiple streaming applications onto many-core SoCs has to provide three key features to the system designer. First, a high-level specification model that hides unnecessary implementation details but provides enough flexibility to specify the dynamic interactions between applications. Second, a tool to calculate an optimal mapping of the applications onto the system in a transparent manner. Third, runtime

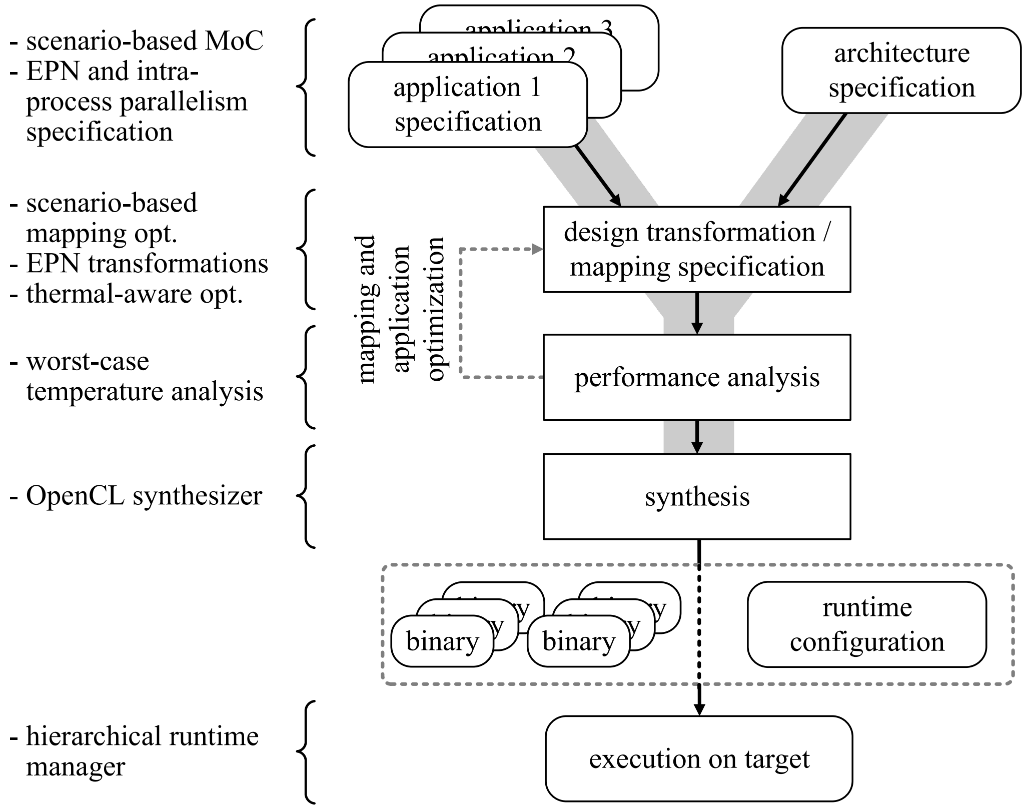

- scenario-based MoC
- EPN and intra-process parallelism specification
- scenario-based mapping opt.
- EPN transformations
- thermal-aware opt.
- worst-case temperature analysis
- OpenCL synthesizer
- hierarchical runtime manager
application 3
application 2
application 1 specification
architecture specification
mapping and application optimization
design transformation / mapping specification
performance analysis
synthesis
binary
binary
runtime configuration
execution on target

Fig. 1.3: Design flow to develop reliable and efficient many-core SoCs. The main contributions of this thesis are listed on the left side of the figure.

support to dynamically change the functionality of the system. The first chapter of this thesis proposes such a design flow. Multi-application support is provided by representing predefined sets of running applications as execution scenarios [GPH$^+$09, SGB10, CLA13] and specifying the interactions between the applications. The proposed design flow contributes to the state-of-the-art in the following ways:

- A scenario-based model of computation for streaming applications is formally described. While each application is separately specified as a KPN, a finite state machine is elaborated to describe the interactions between the different applications.

- We propose a novel hybrid design time and runtime strategy for mapping multiple applications specified by the scenario-based model of computation onto heterogeneous many-core SoCs. We prove that the approach leads to a scalable mapping solution suited for the design of complex embedded systems.

- We formally describe a hierarchically organized runtime manager that is able to handle behavioral and architectural dynamism of many-core SoCs. Failures of the platform are handled by allocating spare cores at design time. We show that this approach is able to tolerate faulty cores without additional design time analysis such that a high responsiveness to faults is achieved.

Chapter 3: Expandable Process Networks

In Chapter 3, we argue that for a certain class of applications, namely applications that are specified as process networks, the application can be specified in a manner that enables automatic exploration of task, data, and pipeline parallelism [GTA06, YH09]. To this end, we propose a formal extension for streaming programming models called Expandable Process Networks (EPNs) that abstracts several possible granularities in a single specification. The EPN semantics facilitates the synthesis of multiple design implementations that are all derived from the same high-level specification. This enables the runtime manager to select the best fitting implementation for the given computing resources thereby minimizing inter-process communication and scheduling overheads. More detailed, the contributions of Chapter 3 are as follows:

- We formally describe the proposed semantics of EPNs. In fact, an application specified as an EPN has a top-level process network that can be refined by hierarchically replacing stateful processes by other process networks.

- We show that the semantics of EPNs enables the automatic exploration of task, data, and pipeline parallelism by means of two design transformations, namely *replication* and *unfolding*. Replicating processes increases data parallelism and structural unfolding of a process increases the task and pipeline parallelism by hierarchically instantiating more processes in the process network. A particular highlight of the proposed semantics is the ability to implicitly specify recursive dependencies.

- An analytic performance model to analyze applications specified as EPNs is proposed and employed to explore different degrees of parallelism.

- In order to react to runtime resource variations, we propose a novel technique to transform an application from one design implementation into an alternative design implementation without discarding the program state of the application.

Chapter 4: Exploiting Multi-Level Hardware Parallelism

Chapter 4 proposes an approach to leverage the different kinds of application parallelism to efficiently exploit the different levels of hardware parallelism offered by heterogeneous many-core SoCs. Pipeline and task parallelism are used to distribute the application to the different processing cores and data parallelism is used to exploit the parallelism offered by an individual core. In order to demonstrate the proposed approach, we extend the previously proposed design flow with the ability to run applications on top of Open Computing Language (OpenCL)-capable platforms. The contributions of this chapter are as follows:

- A systematic approach to exploit the different levels of hardware parallelism offered by heterogeneous many-core systems is described. In particular, the multi-level parallelism offered by an individual core is exploited by executing independent sequential process iterations simultaneously and by calculating independent output tokens in parallel, thereby achieving Single Instruction, Multiple Data (SIMD) parallelism.

- The high-level specification described in Chapter 2 is refined to explicitly specify different kinds of application parallelism. Based on that, a software synthesis tool is implemented to automatically generate OpenCL-capable code.

- A highly efficient runtime-system to execute streaming applications on OpenCL-capable platforms is described. The runtime-system optimizes memory transfers between different cores, thereby reducing the communication overhead automatically. Seamless integration of input/output operations is provided by the ability to execute individual processes as native threads in the context of the operating system.

Chapter 5: Thermal-Aware System Design

Chapter 5 explores thermal-aware optimization strategies to optimize a system design with respect to both performance and temperature, leading to major benefits in terms of a guaranteed and predictable high performance. The timing and thermal characteristics of mapping candidates are evaluated by means of formal worst-case real-time analysis methods to provide safe bounds on the execution time and the maximum chip temperature. The proposed methods are implemented as an extension of the MPA framework [WTVL06] and integrated into the previously proposed design flow to automatically analyze the system in terms of temperature. The contributions of this chapter are as follows:

- A novel technique to calculate a non-trivial upper bound on the peak temperature of a many-core SoC is formally derived and applied to generate a unique thermal-aware optimization framework for many-core SoCs. The aim of the optimization framework is to minimize the worst-case chip temperature of the system while guaranteeing the system's real-time requirements by optimizing the assignment of processes to cores.

- A two-step approach to integrate the proposed formal timing and thermal analysis methods into design space exploration is described. First, the considered timing and thermal analysis models are generated from the same set of specifications as used for software synthesis. Afterwards, the analysis models are calibrated with performance data reflecting the execution of the system on the target platform.

- The viability of the thermal-aware optimization framework is demonstrated by integrating it into the proposed design flow for mapping streaming applications onto many-core SoCs. We validate the proposed techniques in various case studies targeting the Multiprocessor ARM (MPARM) virtual platform [BBB+05]. We demonstrate that there is no single solution that maximizes the performance and minimizes the peak temperature of the system.

2

Scenario-Based System Design

2.1 Introduction

Embedded systems are becoming multi-functional by running multiple applications in parallel. Computing systems embedded in autonomous cars, for instance, will have to solve a multitude of multi-sensorial data fusion and artificial intelligent problems in parallel. Moreover, the functionality of the system can change over time so that embedded systems must be able to deal with complex dynamic interactions between the applications.

Fortunately, by the nature of embedded systems, the use cases of a system, and thus also the applications that may run on the system, are known at design time so that the multi-functional behavior can be modeled as a set of execution scenarios. In fact, each execution scenario then represents a certain use case of the system and is characterized by a set of concurrently running or paused applications. Furthermore, the interactions between the applications can be specified by transitions between the execution scenarios.

To efficiently utilize massively parallel many-core architectures, the system must be able to exploit the available computing resources in each execution scenario separately. However, as state-of-the-art design flows for parallel embedded systems assign the computing resources statically to the applications, they can deal with dynamic behavior only by over-provisioning the system. Therefore, a design flow for mapping dynamically interacting applications onto many-core systems must provide three key features to the system designer:

1. A high-level specification model that hides unnecessary implementation details, but provides enough flexibility to specify dynamic interactions between applications.
2. A tool to calculate an optimal mapping of the applications onto the architecture in a transparent manner.
3. Runtime support to change the functionality of the system.

This chapter proposes the Distributed Application Layer (DAL), a scenario-based design flow for the model-driven development [Sel03] of heterogeneous many-core SoCs. It supports the design, the optimization, and the simultaneous execution of multiple dynamically interacting streaming applications on heterogeneous many-core SoCs. DAL itself is available online for download at `http://www.dal.ethz.ch` and is the basis of this thesis. The methods proposed in conjunction with this thesis are integrated into this design flow.

The DAL design flow has been applied successfully in various projects at ETH Zurich. For instance, in the context of the EU FP 7 project "European Reference Tiled Architecture Experiment" (EURETILE) [PBG$^+$13], which deals with the design and implementation of fault-tolerant many-tile systems, the DAL design flow has been used as front-end to design and optimize many-tile systems with up to 200 tiles.

Overview

The overall structure of the DAL design flow is illustrated in Fig. 2.1. The design flow's input consists of a set of applications, the execution scenarios, and an abstract specification of the architecture. Applications are specified as Kahn Process Networks (KPNs) [Kah74]. As KPNs are determinate, provide asynchronous execution, and are capable to describe data-dependent behavior, they are well suited as basis for a high-level programming model. In case a higher predictability is required, the application model can be restricted, e.g., to Synchronous Dataflow (SDF) graphs [LM87]. To represent the interactions between the applications, a finite state machine is used, where each scenario is represented by a state. Transitions between scenarios are triggered by behavioral events generated by either running applications or the runtime-system.

During design time analysis, a set of optimal mappings is calculated. At runtime, the runtime manager monitors behavioral events, and applies the pre-calculated mappings to start, stop, resume, and pause applications according to the finite state machine. As the number of execution scenarios is restricted, an optimal mapping can be calculated for each execution scenario. However, assigning each execution scenario an individual

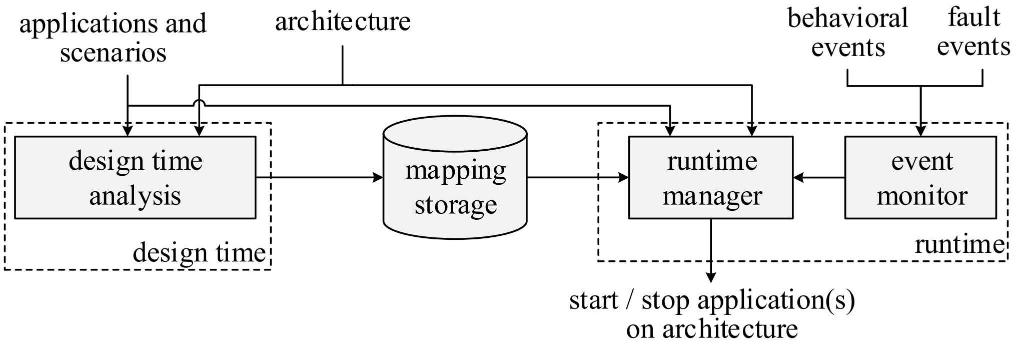


Fig. 2.1: Overall structure of the Distributed Application Layer (DAL) design flow.

mapping might lead to bad performance due to runtime reconfiguration overhead. Therefore, processes are assumed resident, i.e., an application has the same mapping in two connected execution scenarios. The result of this approach is a scalable mapping solution where each application has assigned a set of mappings and each mapping is individually valid for a subset of execution scenarios.

The runtime manager is made up of hierarchically organized controllers that follow the architecture structure and handle the behavioral and architectural dynamism. In particular, behavioral dynamism leads to transitions between execution scenarios and architectural dynamism is caused by temporary or permanent failures on the hardware platform. The controllers monitor behavioral events, change the current execution scenario, and start, stop, resume, or pause certain applications. Whenever they start an application, they select the mapping assigned to the new execution scenario. To handle failures of the platform, spare cores are allocated at design time so that the runtime manager has the ability to move all processes assigned to a faulty physical core to a spare core. As no additional design time analysis is necessary, the approach leads to a high responsiveness to faults.

Outline

The remainder of the chapter is organized as follows: First, related work is discussed. Afterwards, the considered class of many-core SoC platforms is discussed in Section 2.3. In Section 2.4, the proposed model of computation is detailed. In Section 2.5, the hybrid design time and runtime mapping optimization strategy is proposed. In Section 2.6, an exemplified implementation of DAL targeting Intel's Single-chip Cloud Computer (SCC) processor [HDH+10] is described. Finally, in Section 2.7, case studies are shown to illustrate the presented concepts.

2.2 Related Work

Programming paradigms for many-core systems have to tackle various new challenges. Techniques that worked well for systems with just a few cores will become the bottleneck in the next few years [Vaj11]. First case studies about programming many-core systems have shown that productive parallel programming is possible when advanced communication protocols are used [vdWMH11] and the platform is abstracted as a parallel computer architecture with distributed memory [MCP$^+$12]. This motivates the use of a disciplined design methodology to program concurrent applications on complex many-core SoCs. In this section, we discuss various approaches to design multi-functional embedded systems. For an overview of embedded system design flows using the KPN or SDF model of computation, we refer to Section 1.2.

To capture the increasing dynamism in future embedded systems, mapping strategies that generate a set of mappings at design time have been proposed, e.g., in [GPH$^+$09, MAV$^+$10, SGB10]. Then, a runtime mechanism selects the best fitting mapping depending on the actual resource requirements of all active applications. In [GPH$^+$09], the concept of system scenarios is introduced, which divides the system behavior into groups that are similar from a cost perspective. It has been applied in [SGB10] to comprehend the dynamic behavior of an application as a set of scenarios. However, each scenario is specified as a single SDF [LM87] graph. In contrast, our work just specifies the running and paused applications per scenario, and each application is specified as a separate KPN. We think that the KPN model of computation is better suited for high-level programming models and the individual specification of each application enables a better resource usage. Finally, the approach proposed in [MAV$^+$10] generates multiple mappings, which differ from each other in terms of power consumption and performance, at design time. However, the approach is not scalable due to the centralized runtime manager.

The concept of hybrid mapping strategies has already been investigated in various other works. In [BBM08], it is proposed to compute various system configurations and to calculate an optimal task allocation and scheduling for each of them. At runtime, the decision whether a transition between allocations is feasible, is based on pre-calculated migration costs. In our work, we assume that processes are resident. This makes design time analysis more complex, but eliminates undesired disruption caused by process migration. Similarly, process migration is prohibited in [SCT10]. They use statistical methods to compute different mappings for different interconnected usage-scenarios. As the approach evaluates a large number of mappings, it might not scale with the size of the platform.

In [SKS11], a hybrid mapping strategy, which calculates several resource-throughput trade-off points at design time, is proposed. At runtime, it selects the best point with respect to the available resources. However, the approach is restricted to homogeneous platforms and the schedulability of the system is only known at runtime.

In order to tolerate runtime processor failures, a multi-step mapping strategy is proposed in [LKP+10]. After calculating a static mapping for all possible failure scenarios, a processor-to-processor mapping is performed at runtime. As analyzing and storing a mapping scenario for each failure scenario is not scalable in general, we allocate spare architectural units at design time and use them as target for process migration after the occurrence of a fault. This allows us to limit the number of possible failure scenarios so that all of them can be evaluated during design space exploration.

Various options to design a runtime manager have been discussed in literature. On the one hand, a fully centralized approach can be seen as a broker running on its own core. While centralized approaches are widely used in multi-core systems [SLS07, MAV+10], they impose a performance bottleneck on many-core systems. On the other hand, a fully distributed approach [CJS+02, KBL+11] leads to a high complexity. Therefore, we propose a hierarchical centralized approach, that takes system scalability into account at a low complexity.

The KPN model of computation has been extended in [GB04] with the ability to support sporadic control events. However, the proposed Reactive Process Network (RPN) model of computation does neither include a concrete execution semantics nor mapping strategies. By specifying the execution scenarios as a finite state machine, we are able to formally define an execution semantics and to propose a hybrid design time and runtime mapping strategy to efficiently execute multiple dynamically interacting KPNs on a many-core platform. Finally, we define the semantics of a scenario change and propose a high-level programming interface for behavioral and fault events.

2.3 Architecture Model

The input to the DAL design flow is a set of applications and an abstract specification of the architecture. In this section, we discuss the abstract representation of the architecture used to specify many-core SoC platforms in DAL.

We start with the formal definition of the architecture, which we represent as a set of processing cores that are connected by a hierarchically organized set of networks.

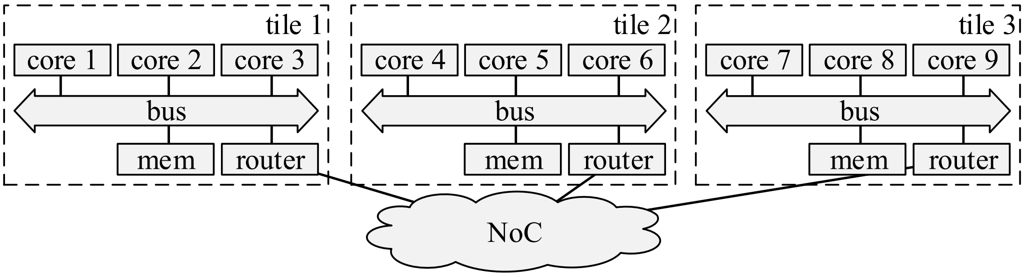


Fig. 2.2: Sketch of a hierarchically organized many-core SoC platform.

Def. 2.1: **(Hierarchically Organized Many-Core Architecture)** *A hierarchically organized many-core architecture* $\mathcal{A} = \langle C, D, N^{(1)}, \ldots, N^{(\eta)}, z \rangle$ *is defined as a set of cores* C*, a set of core types* D*,* η *sets of networks* $N^{(1)}$ *to* $N^{(\eta)}$*, and a function* $z : C \rightarrow D$ *that assigns each core* $c \in C$ *its type* $z(c) \in D$*.*

The set of core types might be used to differ between Digital Signal Processor (DSP) and Reduced Instruction Set Computer (RISC) components or to distinguish between different operation frequencies. Each set of networks corresponds to a communication layer so that the architecture consists of η communication layers. A network $n^{(k)} \in N^{(k)}$ is defined as a subset of C. In particular, each network $n^{(1)} \in N^{(1)}$ represents the intra-core communication, i.e., $|N^{(1)}|=|C|$, and for each $c \in C$, there is a network $n^{(1)} \in N^{(1)}$ with $n^{(1)} = \{c\}$. The second set of networks $N^{(2)}$ partitions the cores into tiles so that each core is assigned to exactly one tile, i.e., we have $\bigcup_{n^{(2)} \in N^{(2)}} = C$ and $n_i^{(2)} \cap n_j^{(2)} = \emptyset$ for all $n_i^{(2)}, n_j^{(2)} \in N^{(2)}$ and $i \neq j$. Similarly, every other set of networks $N^{(k)}$ partitions the cores so that each network $n^{(k)} \in N^{(k)}$ contains multiple subordinate networks, i.e., there exists a network $n^{(k)} \in N^{(k)}$ with $n^{(k-1)} \subseteq n^{(k)}$ for all $n^{(k-1)} \in N^{(k-1)}$, $\bigcup_{n^{(k)} \in N^{(k)}} = C$, and $n_i^{(k)} \cap n_j^{(k)} = \emptyset$ for all $n_i^{(k)}, n_j^{(k)} \in N^{(k)}$ and $i \neq j$. Finally, $N^{(\eta)} = \{n^{(\eta)}\}$ contains a single network hierarchically connecting all processing cores, i.e., $n^{(\eta)} = C$. For notation simplicity, we define the type of a network as the concatenation of all core types of the network.

The hierarchical representation of the architecture is a generalization of the well-known tile-based multiprocessor model [CSG99] that has been applied successfully in academia and industry (e.g., [PJL⁺06, SBGC07, HGBH09]). First prototypes of future many-core SoCs typically consist of three sets of networks, i.e., $\eta = 3$. These sets then correspond to the three communication layers intra-core, intra-tile, and inter-tile communication [VHR⁺08, HDH⁺10, MBF⁺12]. A shared bus is often used for intra-tile communication and a NoC for inter-tile communication. Figure 2.2 sketches a typical many-core SoC platform with $\eta = 3$ and its abstract representation is illustrated in Fig. 2.3.

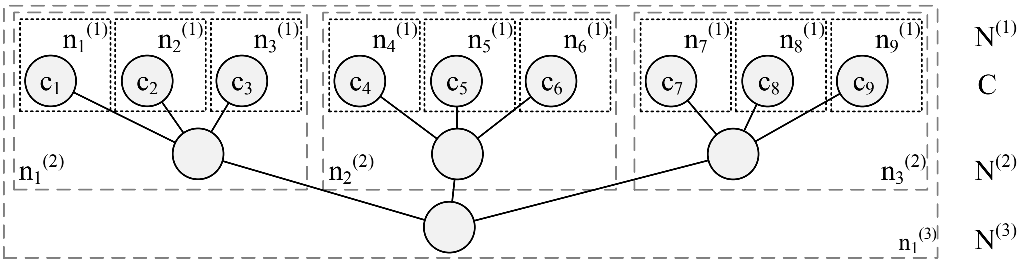


Fig. 2.3: Abstract representation of the hierarchically organized many-core SoC platform sketched in Fig. 2.2.

Due to high power densities, many-core SoCs are prone to various kinds of failures. In this chapter, we restrict ourselves to a failure of a core or a router. In particular, in case that a router fails, we assume that the tile is not anymore available. In any case, we suppose that either the failed component or any other component detects the failure and sends a fault event to the runtime manager. For instance, as part of the EURETILE project, a novel hardware design paradigm named LO|FA|MO [ABF+13] has been developed, which adds an additional fault monitor to each tile. The basic idea of the LO|FA|MO fault detection mechanism is that a software component running on the tile must periodically update its watchdog register. In case the software component misses to update its watchdog register, e.g., due to a fault, the hardware-based fault monitor on the same tile becomes aware of the fault and stops sending diagnostic messages via the network towards its neighbors. When the neighbors do not retrieve these messages anymore, they propagate the information about the faulty tile along the system hierarchy to the runtime manager.

To include the evaluation of all possible failure scenarios in the design time analysis, spare cores and tiles are allocated at design time. We call the abstract representation of the architecture without spare cores and tiles virtual architecture.

Def. 2.2: **(Virtual Architecture)** *A virtual architecture* ${}^{\mathcal{V}}\!\mathcal{A} = \langle {}^{\mathcal{V}}C, D, {}^{\mathcal{V}}N^{(1)}, \ldots, {}^{\mathcal{V}}N^{(\eta)}, {}^{v}z \rangle$ *consists of the set of virtual cores* ${}^{\mathcal{V}}C$*, the set of core types* D *of architecture* $\mathcal{A}$*,* η *sets of virtual networks* ${}^{\mathcal{V}}N^{(1)}$ *to* ${}^{\mathcal{V}}N^{(\eta)}$*, and function* ${}^{v}z$*. The function* ${}^{v}z : {}^{\mathcal{V}}C \rightarrow D$ *assigns each core* ${}^{v}c \in {}^{\mathcal{V}}C$ *its type* ${}^{v}z({}^{v}c) \in D$*.*

Typically, it is the task of the system designer to specify the spare components. One possibility to generate ${}^{\mathcal{V}}\!\mathcal{A}$ is to remove from each network $n^{(i)} \in N^{(i)}$ one subordinate network $n^{(i-1)} \in N^{(i-1)}$ per network type so that each network is able to handle one failure. Finally, each virtual network ${}^{v}n^{(i-i)}$ can be mapped onto any physical network $n^{(i-1)}$ that belongs to the same superior network $n^{(i)}$ and has the same type as

${}^{v}n^{(i-1)}$. For notation simplicity, we will call a physical network with no corresponding virtual representation a spare network and we will use the expression spare network instead of spare core and tile.

Ex. 2.1: *Consider the system illustrated in Fig. 2.3. Suppose that all cores are of the same type and the system designer selects* $n_3^{(1)}$ *as spare network of* $n_1^{(2)}$. *Then the virtual networks* ${}^{v}n_1^{(1)}$ *and* ${}^{v}n_2^{(1)}$ *can be mapped onto the physical networks* $n_1^{(1)}$, $n_2^{(1)}$, *and* $n_3^{(1)}$, *but not on* $n_4^{(1)}$ *as it belongs to a different tile.*

2.4 Scenario-Based Model of Computation

In this section, we formally define the scenario-based model of computation for streaming applications. We first discuss the specification of the individual applications as KPNs. Afterwards, we formalize the dynamic behavior of the system by a set of execution scenarios. Finally, we discuss the execution semantics of the system.

2.4.1 Application Specification

The KPN [Kah74] model of computation is considered to specify the application behavior. In the following, we formally specify an application as a KPN.

Def. 2.3: **(Kahn Process Network)** *A Kahn Process Network (KPN)* $p = \langle V, Q \rangle$ *consists of a set of autonomous processes* $v \in V$ *that communicate through unbounded point-to-point FIFO channels* $q \in Q$ *by reading tokens from an input channel or writing tokens to an output channel. Each process* $v \in V$ *represents a monotonic and determinate mapping* F *from one (or more) input streams to one (or more) output streams whereby monotonicity is obtained by having blocking and destructive read access to the channels.*

Figure 2.4 shows a KPN with four processes and four FIFO channels. As every process $v \in V$ is monotonic and determinate, there is no notion of time and the output just depends on the sequence of tokens in the individual input streams [GB03]. In other words, the output stream $[y_1, y_2, \ldots]$ can be constructed by iteratively applying mapping F to subsequent parts of the corresponding input stream(s) $[x_1, x_2, \ldots]$, and concatenating the results: $\mathbf{Y} = F(\mathbf{X}) : [y_1, y_2, \ldots] = [F(x_1), F(x_2), \ldots]$.

A KPN assumes unbounded point-to-point FIFO channels even though such FIFO channels cannot be realized in a physical implementation. Still, an implementation with the same semantics can represent channels with finite capacity that are accessed using blocking read and

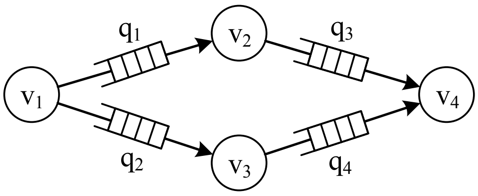


Fig. 2.4: Example of a KPN $p = \langle V, Q \rangle$ with four processes v_1, v_2, v_3, and v_4 and four FIFO channels q_1, q_2, q_3, and q_4.

write functions [BH01, GB03]. Blocking means that a process stalls if it attempts to write to a full FIFO channel or read from an empty FIFO channel.

Conceptually, a KPN is non-terminating, i.e., once the process network has started it does not stop running. However, in order to specify an application that can possibly stop or pause as a KPN, we extend the definition of a KPN with the ability to terminate and pause. To this end, we first propose the high-level Application-Programming Interface (API) illustrated in Listing 2.1 to specify KPN processes. Roughly speaking, the INIT procedure is responsible for the initialization and is executed once at the startup of the application. Afterwards, the execution of a process is split into individual executions of the imperative FIRE procedure, which is repeatedly invoked. Once an application is stopped, the FINISH procedure is called for cleanup. Communication is enabled by calling high-level READ and WRITE procedures and each process has the ability to request a scenario change by calling the SEND_EVENT procedure. Each process has an internal state (represented by the PROCESSDATA structure), which can be used to preserve data among multiple executions of the FIRE procedure. In addition to the functionality of the individual processes, which is specified in C or C++, the topology of the KPN is specified in an XML format, see Listing 2.2.

Now, we are able to introduce and specify the four generic actions START, STOP, PAUSE, and RESUME of a KPN $p = \langle V, Q \rangle$. The semantic of those four actions is summarized in Table 2.1. As stopping an application p might be problematic, the FIRE procedure of all processes $v \in V$ is aborted at predefined points only, such as when process v is calling READ or WRITE, or when the execution of the FIRE procedure is finished. In the case that a process is blocked, i.e., the process attempts to read from an empty channel, the blocking is resolved before the FIRE procedure is aborted. Finally, the FINISH procedure is executed to perform cleanup operations including reading the left over content from the incoming channels or finishing input and output operations.

List. 2.1: Implementation of a KPN process using the proposed API. Note that no assumptions are made regarding the sequence of read and write operations.

```
void INIT(ProcessData *p) {// process initialization
  initialize();
}

void FIRE(ProcessData *p) { // process execution
  READ(PORT_in, buffer, size); // read from fifo
  if (buffer[0] == eventkey) {
    SEND_EVENT(e); // send event e to runtime manager
  }
  manipulate();
  WRITE(PORT_out, buffer, size); // write to fifo
}

void FINISH(ProcessData *p) { // process cleanup
  cleanup();
}
```

List. 2.2: Specification of a KPN with two processes and one channel.

```
<processnetwork>

  <!-- specification of the processes -->
  <process name="producer">
    <port type="output" name="out"/>
    <source type="c" location="producer.c"/>
  </process>

  <process name="consumer">
    <port type="input" name="in"/>
    <source type="c" location="consumer.c"/>
  </process>

  <!-- specification of the channels -->
  <channel capacity="8" name="channel">
    <sender process="producer" port="out"/>
    <receiver process="consumer" port="in"/>
  </channel>

</processnetwork>
```

Tab. 2.1: Description of the four generic action types of a KPN $p = \langle V, Q \rangle$.

action	**description**
START	All processes $v \in V$ and all FIFO channels $q \in Q$ are installed, and the INIT procedure of all processes $v \in V$ is executed once. Afterwards, all processes $v \in V$ are started and the FIRE procedure is continuously called by the system scheduler.
STOP	The FIRE procedure of all processes $v \in V$ is aborted and the FINISH procedure of all processes $v \in V$ is executed. Afterwards, all processes $v \in V$ and all FIFO channels $q \in Q$ are removed.
PAUSE	The FIRE procedure of all precesses $v \in V$ is interrupted and all processes $v \in V$ are temporary detached from the system scheduler.
RESUME	All processes $v \in V$ are restarted and the FIRE procedure is continuously called by the system scheduler.

2.4.2 Execution Scenario Specification

The dynamic behavior of the system is captured by a set of execution scenarios. Each execution scenario represents a set of concurrently running or paused applications. Execution scenario transitions are triggered by behavioral events generated by either running applications, the runtime-system, or the operating system.

Ex. 2.2: *Consider the (simplified) car entertainment system shown in Fig. 2.5. The system has five execution scenarios with one to three applications. After startup, the system enters the '*map*' scenario where the* `MAP` *application is running and displaying the current position of the car on a map. Depending on the situation, the execution scenario might change. For example, the driver starts to drive backwards so that the parking assistant is started (scenario '*parking*'), the voice navigation notifies the driver to take the next exit (scenario '*nav*'), or the driver starts listening to some music (scenario '*map and music*'). In addition, the voice navigation might notify the driver to change the driving direction while listening to music. To this end, the system switches to scenario '*nav and music*', and pauses the* `MP3` *application.*

In the following, we formally specify the above described dynamic behavior of a many-core system by a finite state machine.

Def. 2.4: **(Dynamic System Behavior)** *The dynamic behavior of a many-core system is represented by a finite state machine* $\mathcal{F} = \langle S, E, T, P, s_0, a, r, h \rangle$ *that consists of the set of execution scenarios* S*, the set of events* E*, the set of directed transitions* $T \in S \times S$*, the set of applications* P*, an initial execution scenario* $s_0 \in S$*, and three functions* a*,* r*, and* h*. The function* $a : T \to E$ *maps a transition* $t \in T$ *to a set of triggering events* $a(t) \subseteq E$ *for all* $t \in T$*. The function* $r : S \to P$ *assigns each*

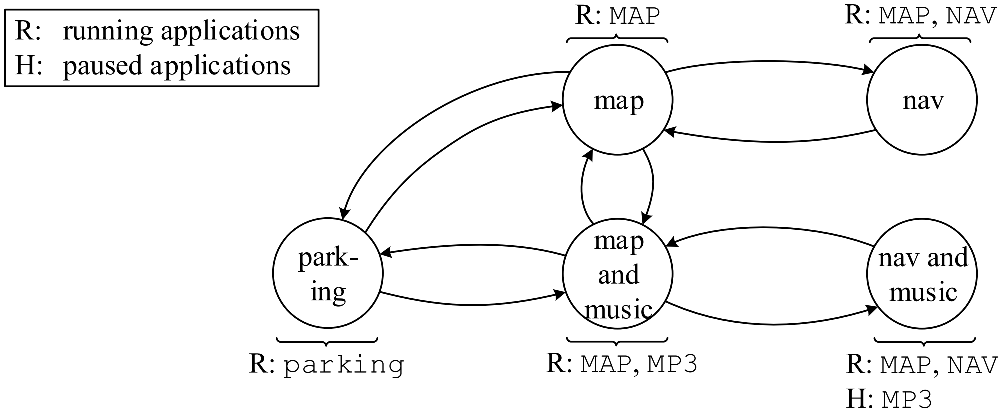


Fig. 2.5: Exemplified specification of a (simplified) car entertainment system that consists of five execution scenarios with one to three applications.

execution scenario $s \in S$ *a set of running applications* $r(s) \subseteq P$ *and the function* $h : S \rightarrow P$ *assigns each execution scenario* $s \in S$ *a set of paused applications* $h(s) \subseteq P$. *We suppose that there is only one instance of an application so that* $r(s) \cap h(s) = \emptyset$ *for all* $s \in S$.

Ex. 2.3: *Figure 2.6 presents an example of a finite state machine* $\mathcal{F} = \langle S, T, E, P, s_0, a, r, h \rangle$ *with four execution scenarios* s_0, s_1, s_2, *and* s_3 *among which* s_0 *is initially active. The execution scenarios are linked by the set of transitions* $T = \{t_1, t_2, t_3, t_4, t_5\}$ *such that* $t_1 = \langle s_0, s_1 \rangle$, $t_2 = \langle s_1, s_2 \rangle$, $t_3 = \langle s_2, s_3 \rangle$, $t_4 = \langle s_3, s_1 \rangle$, *and* $t_5 = \langle s_3, s_0 \rangle$. *Function a assigns each transition its triggering events. For example, transition* t_2 *from execution scenario* s_1 *to execution scenario* s_2 *happens when the events* e_2 *or* e_3 *are detected in execution scenario* s_1. *Finally, functions r and h assign each execution scenario a list of running and paused applications.*

2.4.3 Execution Semantics

The above introduced model of a finite state machine $\mathcal{F}$ is a Moore machine [Moo56], i.e., each execution scenario has a list of running and paused applications, and each transition between execution scenarios has a set of events that trigger the transition. However, in terms of execution, each transition is associated with a set of actions. For example, transition t_1 of the finite state machine $\mathcal{F}$ illustrated in Fig. 2.6 is associated with the action {*pause application* p_1}, and transition t_4 is associated with the actions {*stop application* p_3, *start application* p_2}.

Therefore, in terms of execution, we map the system evolution to a Mealy machine [Mea55] and transform $\mathcal{F}$ into a new finite state machine.

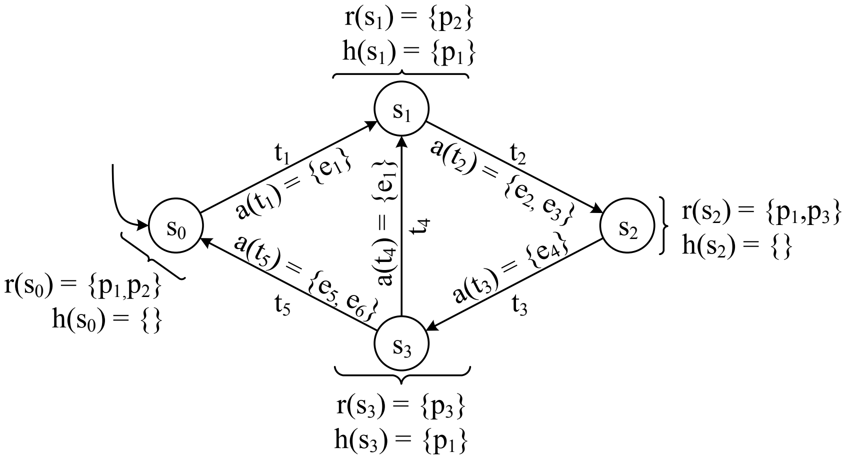


Fig. 2.6: Example of a finite state machine $\mathcal{F} = \langle S, T, E, P, s_0, a, r, h \rangle$.

Def. 2.5: **(Execution Semantics)** *The execution semantics of a many-core SoC is represented by the finite state machine* $\widetilde{\mathcal{F}} = \langle S, E, \widetilde{T}, P, t_0, a, u^{\mathrm{s}}, u^{\mathrm{t}}, u^{\mathrm{p}}, u^{\mathrm{r}} \rangle$ *that consists of the set of execution scenarios* S*, the set of events* E*, the set of directed transitions* $\widetilde{T} \in S \times S$*, the set of applications* P*, an initial transition* $t_0 \in T$*, and six functions* a*,* u^{s}*,* u^{t}*,* u^{p}*, and* u^{r}*.* S*,* E*,* P*, and* $a : T \rightarrow E$ *are defined as in Definition 2.4 and* $\widetilde{T} = T \cup t_0$*. The functions* u^{s}*,* u^{t}*,* u^{p}*, and* u^{r} *assign each transition* $t \in \widetilde{T}$ *the set of applications to be started, stopped, paused, and resumed. In particular, suppose that transition* $t = \langle s_x, s_y \rangle$*, then* $u^{\mathrm{s}}(t)$*,* $u^{\mathrm{t}}(t)$*,* $u^{\mathrm{p}}(t)$*, and* $u^{\mathrm{r}}(t)$ *are defined by:*

$$\begin{array}{llcll}
\textsc{start}: & u^{\mathrm{s}}(t) & = & r(s_y) \setminus (r(s_x) \cup h(s_x)) & \subseteq P \\
\textsc{stop}: & u^{\mathrm{t}}(t) & = & (r(s_x) \cup h(s_x)) \setminus (r(s_y) \cup h(s_y)) & \subseteq P \\
\textsc{pause}: & u^{\mathrm{p}}(t) & = & h(s_y) \cap r(s_x) & \subseteq P \\
\textsc{resume}: & u^{\mathrm{r}}(t) & = & r(s_y) \cap h(s_x) & \subseteq P
\end{array}$$

In other words, whenever transition $t \in \widetilde{T}$ is triggered, all applications $p \in u^{\mathrm{s}}(t)$ are started, all applications $p \in u^{\mathrm{t}}(t)$ are stopped, all applications $p \in u^{\mathrm{p}}(t)$ are paused, and all applications $p \in u^{\mathrm{r}}(t)$ are resumed.

In terms of execution, the initial transition t_0 takes place after startup so that the finite state machine enters execution scenario s_0. Whenever an event $e \in E$ that corresponds to one of the outgoing transitions of the current execution scenario, is received, the transition takes place. In other words, an event $e \in E$ triggers a transition $t \in T$ if and only if $e \in a(t)$, $t = \langle s_x, s_y \rangle$, and s_x is the current execution scenario of the finite state machine.

Conceptually, the reaction of the system to an event is immediate, i.e., the actions listed in Table 2.1 are performed in zero time. However, as the production and execution of these actions take a certain amount of time, we have to come up with additional rules, which preserve the described semantics. In particular, we insist that a transition is only triggered if the system is in a stable scenario. A stable scenario is reached if the executions of all actions that have been triggered by the previous transition, are completed. This rule is required as events might arrive faster than they can be processed. If the system is not yet in a stable scenario, the execution of new actions might cause the system to move to an unknown or wrong execution scenario. Practically, this requirement can be realized by saving all incoming events in a FIFO queue so that the events are processed in a First-Come-First-Served (FCFS) manner. If the current execution scenario has an outgoing transition that is sensitive to the head event of the FIFO queue, the transition takes place and the event is removed from the FIFO queue. Otherwise, the event is removed without changing the active execution scenario.

2.5 Hybrid Mapping Optimization

In this section, we present a hybrid design time and runtime strategy for mapping streaming applications onto many-core SoC platforms. The design time component calculates an optimal mapping for each application and execution scenario where the application is either running or paused. At runtime, the dynamic mapping of the applications onto the architecture is controlled by a runtime manager, which monitors events, chooses an appropriate mapping, and finally executes the required actions, see Fig. 2.7.

2.5.1 Design Time Analysis and Optimization

In this subsection, we introduce the proposed approach for design time optimization. We minimize the maximum utilization of each core subject to utilization and communication constraints so that we obtain a system with a balanced workload. Minimizing the maximum core utilization enables us to reduce the operation frequency, thereby saving energy, or to increase the invocation interval of the applications to increase the overall throughput.

In order to maximize the performance, a different mapping should be assigned to each execution scenario. However, changing the process-to-core assignment with each scenario transition might lead to bad performance due to reconfiguration overhead. Therefore, the approach proposed in this chapter assumes that processes are resident, i.e., once a process starts its execution on a certain core, it will not be remapped to another core. In other words, if an application is active in two connected

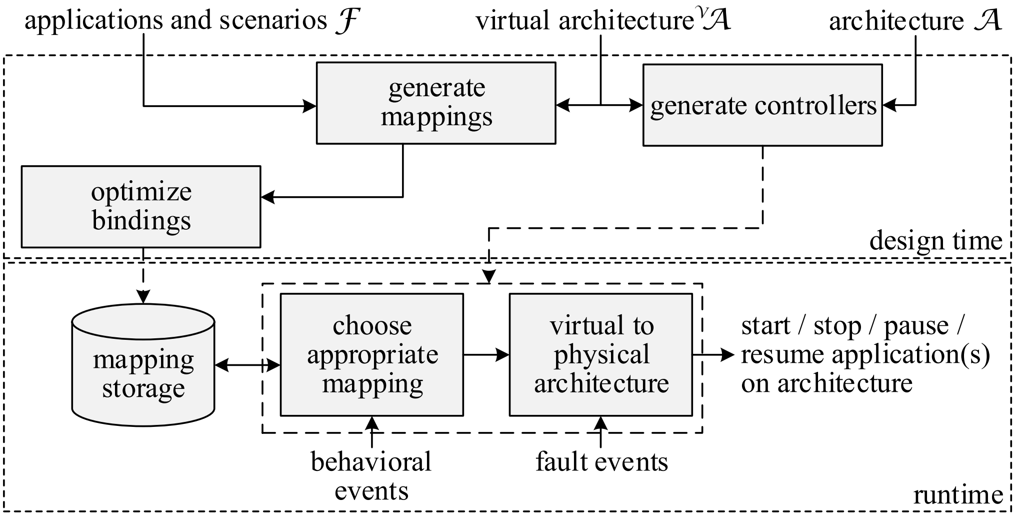

applications and scenarios $\mathcal{F}$
virtual architecture ${}^{\mathcal{V}}\mathcal{A}$
architecture $\mathcal{A}$
generate mappings
generate controllers
optimize bindings
design time
mapping storage
choose appropriate mapping
virtual to physical architecture
start / stop / pause / resume application(s) on architecture
behavioral events
fault events
runtime

Fig. 2.7: Overall hybrid mapping optimization approach with a design time and a runtime component.

execution scenarios, it has the same mapping in both execution scenarios. We think that this restriction is well suited for embedded systems where process migration leads to non-negligible costs in terms of time and system overhead.

Ex. 2.4: *Consider again the car entertainment system outlined in Fig. 2.5. The* `MAP` *application will have the same mapping in all active execution scenarios. However, the mapping of the* `NAV` *application might be different in scenario* 'nav' *and scenario* 'nav and music' *as they are not connected by a direct transition. The runtime manager selects the appropriate mapping for the* `NAV` *application when the application is started depending on the current execution scenario.*

Mapping Specification

The design time component calculates an optimal mapping for each application and execution scenario where the application is either running or paused. Thus, the output of the design time analysis is a collection M of optimal mappings and exactly one mapping $m \in M$ is valid for a pair of application and execution scenario. Formally, we define a mapping as follows.

Def. 2.6: **(Mapping)** *A mapping $m \in M$ is a triple $\langle p, S^m, B^m \rangle$ where p is an application, $S^m \subseteq S$ is a subset of execution scenarios, and $B^m \in V \times {}^{\mathcal{V}}C$ is the set of binding relations. S^m denotes the set of execution scenarios for which mapping m is valid. As processes are resident, the same mapping might be valid for more than one execution scenario. Finally, a binding relation $\langle v, {}^{v}c \rangle \in B^m$ specifies that process v is bound to virtual core ${}^{v}c$.*

In the following, we propose a two-step procedure to calculate mappings $m \in M$. First, we calculate which pairs of application and execution scenario must use the same mapping so that no process migration is required. At the end of the first step, we allocate one mapping $m \in M$ for each of these pairs. Afterwards, in a second step, we calculate for each mapping $m \in M$ a set of optimized binding relations so that the objective function is minimized and additional architectural constraints are fulfilled.

Mapping Generation

First, we calculate pairs $\langle p, S^m \rangle$ of an application p and a subset of execution scenarios S^m so that the size of each subset is minimized and additional constraints are fulfilled. The additional constraints are designed to ensure that process migration is not required, i.e., they guarantee that application p has the same mapping in all execution scenarios $s \in S^m$. In particular, we can identify the following three constraints:
Constraint 1: Each application is mapped:

$$p \in (r(s) \cup h(s)) \quad \Rightarrow \exists\, m = \langle p, S^m, B^m \rangle \in M : s \in S^m. \tag{2.1}$$

Constraint 2: Two mappings do not overlap:

$$m_1 = \langle p, S^{m_1}, B^{m_1} \rangle \text{ and } m_2 = \langle p, S^{m_2}, B^{m_2} \rangle \quad \Rightarrow S^{m_1} \cap S^{m_2} = \emptyset. \tag{2.2}$$

Constraint 3: Process migration is not allowed:

$$\begin{aligned} & p \in ((r(s_1) \cup h(s_1)) \cap (r(s_2) \cup h(s_2))) \text{ and } t = \langle s_1, s_2 \rangle \in T \\ & \Rightarrow \exists\, m = \langle p, S^m, B^m \rangle \in M : s_1, s_2 \in S^m. \end{aligned} \tag{2.3}$$

The mapping generation problem can be solved by calculating for each application p the maximally connected components of a subgraph that just contains all execution scenarios where p is either running or paused. Then, we can generate a new pair $\langle p, S^m \rangle$ for each component of this subgraph. Algorithm 2.1 presents the pseudo code to calculate all pairs $\langle p, S^m \rangle$. The algorithm generates the pairs $\langle p, S^m \rangle$ by sequentially analyzing all applications. First, a subgraph $\mathcal{G} = \langle S^{sub}, T^{sub} \rangle$ is determined by removing all execution scenarios $s \in S$ in which application p is neither running nor paused. Then, the maximally connected components $\mathcal{G}_i^{conn} = \langle S_i^{conn}, T_i^{conn} \rangle \in \mathcal{G}^{conn}$ of subgraph $\mathcal{G}$ are determined. In other words, the execution scenarios are partitioned into non-overlapping sets such that there is no transition between nodes in different subsets $\mathcal{G}_i^{conn}$ and the subsets are as large as possible. Finally, a new pair $\langle p, S_i^{conn} \rangle$ is generated for each maximally connected component of a subgraph

Algorithm 2.1 Pseudo code to generate all pairs $\langle p, S^m \rangle$ of an application p and a subset of execution scenarios S^m so that the number of elements per subset is minimized and the constraints specified by Eqs. (2.1) to (2.3) are fulfilled.

```
Input: finite state machine $\mathcal{F} = \langle S, T, E, P, s_0, a, r, h \rangle$
Output: set of pairs $\langle p, S^m \rangle$
for all applications $p \in P$ do
    $S^{sub} \leftarrow (s \in S | p \in (r(s) \cup h(s)))$.
                                    ▹all scenarios where p is running or paused
    $T^{sub} \leftarrow (t = \langle s_1, s_2 \rangle \in T | p \in (r(s_1) \cup h(s_1))$ and $p \in (r(s_2) \cup h(s_2)))$
                                    ▹all transitions that affect p
    $\mathcal{G} \leftarrow \langle S^{sub}, T^{sub} \rangle$
    $\mathcal{G}^{conn} \leftarrow$ set of all maximally connected components of $\mathcal{G}$
    for all $\mathcal{G}_i^{conn} \in \mathcal{G}^{conn}$ do        ▹generate pairs for each component
        add $\langle p, S_i^{conn} \rangle$        ▹add the new pair to the set of pairs
    end for
end for
```

$\mathcal{G}$. By relying on a breadth-first search algorithm to calculate the set of all maximally connected components, the calculation of all pairs has a computational complexity of $O(|P| \cdot (|T| + |S|))$. In order words, the complexity to calculate all pairs scales linearly with the number of processes and with the total number of execution scenarios and transitions. Typically, the limiting factor is the number of transitions, which can be up to $2 \cdot |S|^2$.

Finally, as application p uses the same mapping in all execution scenarios $s \in S^m$, we allocate a mapping $m = \langle p, S^m, \cdot \rangle \in M$ for each pair $\langle p, S^m \rangle$.

Mapping Optimization

In the second step, we calculate for each mapping $m \in M$ the set of binding relations B^m so that the objective function, i.e., the maximum core utilization, is minimized and predefined architectural constraints are fulfilled. Later, we will use an extended version of the PISA framework [BLTZ03] to solve the considered mapping problem. In particular, the PISA framework uses an Evolutionary Algorithm (EA) to solve the optimization problem.

The number of firings per time unit of process v is $f(v)$ and the maximum execution time of process v on a core of type d is $w(v, d)$. Furthermore, $M^s \subseteq M$ denotes the subset of all mappings with $s \in S$ and $p \in r(s)$, i.e., $M^s = \{\langle p, S^m, B^m \rangle \in M | p \in r(s)$ and $s \in S^m\}$. The binding relations B^m

are calculated so that the maximum core utilization is minimized, and the utilization and communication constraints are met in each execution scenario.

Objective function: The optimization goal of this problem is to minimize the maximum core utilization. In order to incorporate the different execution scenarios into a single objective function, we assign each execution scenario $s \in S$ a weight χ_s [SCT10] so that the object function can formally be stated as:

$$\min\left(\max_{{}^{\mathcal{V}}c \in {}^{\mathcal{V}}C} \sum_{\{s \in S\}} \sum_{\{m \in M^s\}} \sum_{\{v \in V : \langle v, {}^{\mathcal{V}}c \rangle \in B^m\}} \chi_s \cdot f(v) \cdot w(v, {}^{\mathcal{V}}z({}^{\mathcal{V}}c))\right). \tag{2.4}$$

The weight of an execution scenario may represent an importance weighting or an execution probability and can be based on the characteristics of the applications or on historic data.

Constraint 4: In order to make sure that the cores are able to handle the processing load, the following relation has to be satisfied for all cores ${}^{\mathcal{V}}c \in {}^{\mathcal{V}}C$ and states $s \in S$ of finite state machine $\mathcal{F}$:

$$\sum_{\{m \in M^s\}} \sum_{\{v \in V : \langle v, {}^{\mathcal{V}}c \rangle \in B^m\}} f(v) \cdot w(v, {}^{\mathcal{V}}z({}^{\mathcal{V}}c)) \leq 1. \tag{2.5}$$

Constraint 5: Similarly, we can formulate the bandwidth requirement for each network by adding the data volume per time unit of each channel. Then, the aggregated data volume for each network n must be smaller than its supported rate. As the applications are mapped onto a virtual architecture, all possible separations between the processes must be considered. However, due to the hierarchical structure of the architecture, a virtual network is only mapped onto a physical network within the same superior network so that the maximum separation is bounded.

In case that the system can be divided into multiple parts that are behavioral independent, each part can be optimized individually as the following example shows.

Ex. 2.5: *Consider again the car entertainment system illustrated in Fig. 2.5. As scenario* 'parking' *is behavioral independent of all other scenarios, it can be optimized separately.*

2.5.2 Runtime Manager

In this subsection, we discuss the required runtime support to execute a set of applications P on a many-core architecture $\mathcal{A}$. The required runtime support is provided by a runtime manager that has the task to generate commands towards the runtime-system so that the execution semantics described in Section 2.4.3 is ensured. An exemplified runtime-system that provides this functionality for the Intel SCC processor [HDH+10] is described in the next section.

Traditionally, runtime managers are either centralized or distributed. However, as a centralized approach comes with a performance bottleneck and a distributed approach leads to a high complexity, both approaches are not suited for embedded many-core systems. In this thesis, we propose to split the workload among hierarchically organized controllers. In the following, we first discuss the general ideas of a hierarchical control mechanism. Afterwards, we describe the functionalities of the different controllers.

Hierarchical Control Mechanism

The general idea of the hierarchical control mechanism is to assign each network $n \in \{N^{(2)}, \dots, N^{(\eta)}\}$ its own controller $v^c \in V^c$ that handles all intra-network dynamism. In particular, the controller assigned to network $n \in N^{(2)}$ monitors for behavioral and fault events. Whenever such a controller receives an event, it handles the event if it only affects the network of the controller, and otherwise, it sends the event to the controller of its superior network.

By modeling the communication medium between the controllers as FIFO channels, the hierarchical control mechanism can be represented as a process network $p^c = \langle V^c, Q^c \rangle$. To provide bidirectional communication, two FIFO channels are allocated between a controller and its superior controller. Algorithm 2.2 shows the pseudo code to generate process network p^c for an architecture $\mathcal{A}$.

Figure 2.8 shows the process network p^c for the exemplified architecture shown in Fig. 2.3. Following the hierarchical structure of the considered many-core SoC architectures, we categorize the controllers into three different types:

- A SLAVE controller is responsible for a tile, i.e., for a network $n^{(2)} \in N^{(2)}$. All architectural units in network $n^{(2)}$ and all processes v assigned to a core $c \in n^{(2)}$ are able to send events to the SLAVE controller. In addition, a SLAVE controller is able to send commands to the runtime-system of its tile.

Algorithm 2.2 Pseudo code to calculate process network p^c of the hierarchical control mechanism.

```
function ComputeController( arch. $\mathcal{A}$, total number of layers $\eta$ )
    $V \leftarrow$ master
                ▹initial set of controllers that consists of the master controller
    $Q \leftarrow \emptyset$            ▹initial set of FIFO channels connecting the controllers
    ComputeLayer( master, $\eta - 1$, $\mathcal{A}$, $V$, $Q$ )
    $p^c = \langle V, Q \rangle$
    return $p^c$
end function

function ComputeLayer( superior controller $v^p$, layer $l$,
        architecture $\mathcal{A}$, set of processes $V$, set of FIFO channels $Q$ )
    for all $n \in N^{(l)}$ do
        if $l == 2$ then
            $V \leftarrow V \cup$ slave                          ▹add slave controller
        else
            $V \leftarrow V \cup$ interlayer                ▹add interlayer controller
            ComputeLayer( $v$, $l - 1$, $\mathcal{A}$, $V$, $Q$ )
        end if
        $Q \leftarrow Q \cup \langle v, v^p \rangle \cup \langle v^p, v \rangle$        ▹add channels between controllers
    end for
end function
```

- An interlayer controller is responsible for a network $n^{(i)} \in N^{(i)}$ with $i = [3, \eta - 1]$ and η the number of communication layers. It receives all events that cannot be handled by its subordinates. The interlayer controller processes an event if it only affects its own network. Otherwise, it sends the event to its superior controller.

- The master controller is responsible for network $n^{(\eta)} \in N^{(\eta)}$. It processes all events that cannot be handled by any other controller.

Nowadays, all cores of a tile have typically a common operating system so that one slave controller can dynamically allocate processes on all cores of the tile. However, in case that each core has its dedicated runtime-system, we assign a slave controller to each core so that the interaction between the control mechanism and the runtime-system is ensured.

Hierarchical Event Processing

A controller of the hierarchical control mechanism handles the events that only affect the network of the controller. Otherwise, the controller sends

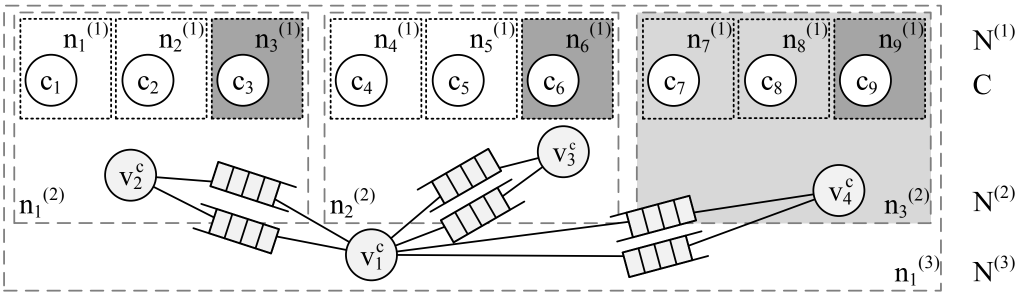


Fig. 2.8: Runtime manager for the architecture illustrated in Fig. 2.3. v^c_2, v^c_3, and v^c_4 are SLAVE controllers and v^c_1 is the MASTER controller. $n^{(1)}_3$, $n^{(1)}_6$, $n^{(1)}_7$, and $n^{(2)}_3$ are networks representing spare networks.

the event to the controller of its superior network. In the following, we detail this procedure for an INTERLAYER controller.

So far, we have seen that events can be categorized into two groups that cause different behavior. The first group contains the behavioral events and the second group contains the fault events.

Def. 2.7: **(Behavioral Event)** *A behavioral event triggers a transition between two execution scenarios and is generated by either running applications or the runtime-system.*

Def. 2.8: **(Fault Event)** *A fault event is triggered in the case that a hardware failure occurs. It is of the form $\langle tag, n\rangle$ whereby tag denotes the fault type and n denotes the affected network.*

Fault events only change the mapping of the virtual architecture ${}^{V}\mathcal{A}$ onto the physical architecture $\mathcal{A}$, but not the mapping of the applications onto the virtual architecture ${}^{V}\mathcal{A}$. Consequently, each controller consists of two components, see Fig. 2.7. The first component is responsible to handle behavioral events and ensures the execution semantics. It is just aware of the virtual architecture ${}^{V}\mathcal{A}$, i.e., it generates commands towards ${}^{V}\mathcal{A}$. The second component processes the fault events and redirects the commands to the corresponding physical network.

Next, we will detail the procedure of an INTERLAYER controller when it receives a fault event. To this end, we suppose that controller v^c belongs to network $n^{(k)}$. Once it receives a fault event of the form $\langle fault, n^{(l)}\rangle$, it executes the procedure outlined in Algorithm 2.3. If $n^{(l)}$ is not a subordinate network of $n^{(k)}$, i.e., $l \neq k-1$, v^c has to reinstall the affected channels (Lines 9–11). Otherwise, if $n^{(l)}$ is a subordinate network of $n^{(k)}$, i.e., $l = k-1$, v^c has to handle the fault by migrating all processes mapped onto the faulty network $n^{(l)}$ to a spare physical network (Lines 2–8).

Algorithm 2.3 Pseudo code to handle a fault event $\langle fault, n^{(l)} \rangle$ under the assumption that controller v^c belongs to network $n^{(k)}$.

```
Input: fault event $\langle fault, n^{(l)} \rangle$
${}^{v}n \leftarrow$ virtual network mapped onto $n^{(l)}$
if $l == k - 1$ then                                          ▷ $n^{(l)}$ is subordinate of $n^{(k)}$
    if $n^{(k)}$ has spare subordinate network $n_s^{(l)}$ of
            the same type as $n^{(l)}$ then                   ▷ $v^c$ handles the fault
        migrate ${}^{v}n$ to $n_s^{(l)}$
    else             ▷ $v^c$ is unable to handle the fault and reports $n^{(k)}$ as faulty
        send $\langle fault, n^{(k)} \rangle$ to superior network and return
    end if
end if
for all $q \in Q$ that connect a process $v_1$ mapped onto ${}^{v}n$ with a process
        $v_2$ mapped onto a physical core $c \in n^{(k)}$ and $c \notin n^{(l)}$ do
    reinstall $q$
end for
if $\exists q \in Q$ that connects a process $v_1$ mapped onto ${}^{v}n$ with a process $v_2$
        mapped onto a physical core $c \notin n^{(k)}$ then
    send $\langle fault, n^{(l)} \rangle$ to superior network
end if
```

Since a fault can be handled without additional mapping optimization, the system has a high responsiveness to faults. In case that network $n^{(l)}$ is not anymore faulty, it sends a reintegration event of the form $\langle available, n^{(l)} \rangle$ to the controller, which marks $n^{(l)}$ as a spare network.

2.6 Exemplified Implementation Targeting the Intel SCC Processor

In the previous sections, the basic concepts to execute multiple streaming applications simultaneously on a many-core system have been presented. To demonstrate the viability of these concepts, a prototype implementation of DAL has been developed targeting Intel's SCC processor [HDH⁺10]. Even though this section focuses on the SCC, the same concepts can be applied to port DAL to any other platform that is running a Linux-based operating system.

Executing applications specified according to the previously proposed scenario-based model of computation on Intel's SCC processor requires a runtime-system and a program synthesis back-end. While the runtime-system provides an implementation of the API, the program synthesis back-end creates the required components to execute the applications on top of the runtime-system. In the following, we first give a high-level overview of the Intel SCC processor. Then, we individually describe the proposed runtime-system and program synthesis back-end.

2.6.1 Intel Single-chip Cloud Computer Processor

The Intel SCC processor [HDH+10] consists of 24 tiles that are organized into a 4×6 grid and linked by a 2D mesh on-chip network. A tile contains a pair of P54C processor cores, a router, and a 16 KB block of SRAM. Each core has an independent L1 and L2 cache. The on-tile SRAM block is also called Message Passing Buffer (MPB) as it enables the exchange of information between cores in the form of messages. The SCC processor is schematically outlined in Fig. 1.1a.

Intel's SCC processor has three communication layers. The first layer is the intra-core communication. The second layer is the intra-tile communication, i.e., every intra-tile network $n^{(2)} \in N^{(2)}$ consists of two cores that communicate without using the on-chip network. Finally, the 24 $n^{(2)}$ networks communicate via the 2D mesh on-chip network and form network $n^{(3)}$. As each tile is only composed of two cores, it might not be appropriate to allocate spare cores, but to reserve one or two tiles as spare tiles.

In terms of programmability, the most commonly used software platform for Intel's SCC processor is to run a Linux kernel on each core. For communication purpose, various message passing libraries have been developed including RCCE [vdWMH11], iRCCE [CLRB11], and RCKMPI [URK11]. Instead of implementing our own software platform, we run our runtime-system on top of the Linux kernel. This enables us to use the basic multi-threading mechanisms provided by the Linux kernel.

2.6.2 Runtime-System

The task of the runtime-system is to provide services for inter-process communication and a mechanism to iteratively execute processes by calling their FIRE procedure. Furthermore, the runtime-system must be able to receive and process the commands of the runtime manager, i.e., it must provide services to manage processes and channels at runtime. In the following, we discuss each of these tasks individually.

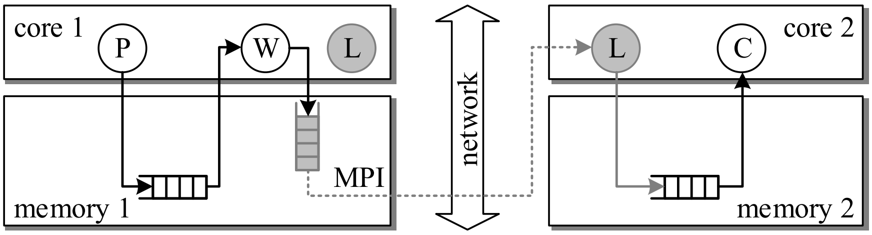


Fig. 2.9: Example of intra- and inter-core communication on the Intel SCC processor. Process 'P' and process 'W' communicate over shared memory and process 'W' and process 'C' communicate over distributed memory. The virtual FIFO and the LISTENER threads are illustrated in grey.

Inter-Process Communication

To be efficient, the runtime-system has to differ between communication over shared and distributed memory, see Fig. 2.9. While ring buffers in private memory are used for intra-core communication, an advanced architecture-dependent FIFO implementation is required for efficient inter-core communication.

In general, FIFO channels might be implemented in private memory of the sender process or receiver process, or in shared memory. We persistently implement the FIFO channels in the private memory of the receiver and use the RCKMPI library [URK11] for inter-core communication. The RCKMPI library is an implementation of the Message Passing Interface (MPI) [Pac96] for the Intel SCC processor. It automatically takes the memory organization of the SCC into account and uses the MPB, if appropriate. As no Direct Memory Access (DMA) controller is available on the SCC for inter-core communication, we launch a LISTENER thread on each core. The LISTENER thread is responsible for handling all incoming traffic and writing the data to the correct FIFO channel. To keep the LISTENER thread lightweight, it uses the memory of the local FIFO channel as receive buffer for data transfer, thereby avoiding expensive allocation and copy operations. Due to the fact that the RCKMPI library is basically a standard implementation of MPI, the discussed implementation of DAL can be ported with low effort to any other Linux platform that supports MPI.

To avoid deadlocks, the LISTENER thread must not be blocked at any time, i.e., we have to ensure that a data transfer is only initialized if the receiver has enough space available to store the data. This is ensured by a virtual FIFO at the sender. The virtual FIFO has the same metadata (amount of free space) as the actual FIFO, but if a process attempts to write, either the data is directly transferred or the calling process is blocked as

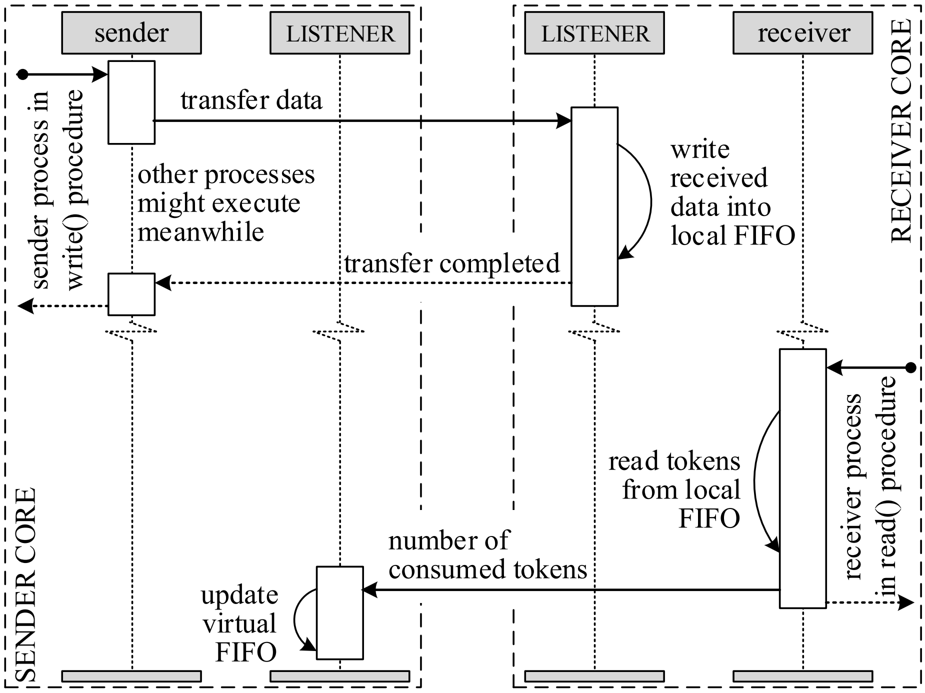

sender
LISTENER
LISTENER
receiver
sender process in write() procedure
transfer data
other processes might execute meanwhile
write received data into local FIFO
transfer completed
RECEIVER CORE
read tokens from local FIFO
receiver process in read() procedure
number of consumed tokens
update virtual FIFO
SENDER CORE

Fig. 2.10: Inter-process communication protocol between processes located on different cores as employed on Intel's SCC processor. Besides the sender and receiver processes, the LISTENER threads of the sender and receiver cores are shown.

long as the receiver has not enough space to store the data. The disadvantage of this approach is that the virtual FIFO has to know when the receiver process has consumed data, thereby requiring synchronization between sender and receiver process. In our implementation, this synchronization is ensured by a signal that is sent by the receiver process when it has consumed data. The inter-process communication protocol is sketched in Fig. 2.10.

Multi-Processing

As our runtime-system runs on top of a Linux kernel, we use the multiprocessing features provided by the operating system to run multiple processes in a quasi-parallel fashion on a single core. In particular, processes are mapped onto Portable Operating System Interface (POSIX) threads and scheduled by the operating system's scheduler. When a process is blocked due to empty input or full output channels, the scheduler automatically selects a different process to execute. Instead of POSIX threads, any other thread implementation could have been used for this task. An overview of possible thread implementations for KPNs is given in [HSH+09].

Process and Channel Management

In order to process the commands of the runtime manager, the runtime-system has to provide services to install, uninstall, start, and stop processes, as well as to create and destroy FIFO channels.

Since the memory footprint of the system is reduced by storing the individual processes as dynamic libraries, the runtime-system must load the dynamic libraries when the application is started. Thus, installing a process involves loading and dynamically linking the corresponding library, and then executing its INIT procedure. Similarly, uninstalling a process involves executing the FINISH procedure and unloading the dynamic library. The procedure to create a channel depends on the mapping of the sender and receiver. If both processes are mapped onto the same core, a local FIFO channel is instantiated. Otherwise, a virtual FIFO channel and a local FIFO channel are instantiated at the sender and receiver, respectively. Afterwards, virtual and local FIFO channels are registered at the corresponding LISTENER thread. A FIFO channel is destroyed by deregistering it at the LISTENER thread and freeing the memory buffer. Finally, a process is started by registering the thread at the scheduler, and stopped by aborting the FIRE procedure and deregistering it from the scheduler.

2.6.3 Program Synthesis Back-end

The program synthesis back-end is the second component required to execute KPNs on top of Intel's SCC processor. The task of the back-end is to create the runtime manager, to embed each process into a POSIX thread, and to create a MAIN function for each core, see Fig. 2.11.

Runtime Manager Synthesis

The task of the runtime manager synthesis is to automatically construct the runtime manager according to the concepts described in Section 2.5.2. With respect to Intel's SCC processor, the result of this step is a process network with one MASTER controller and one SLAVE controller per core. The MASTER controller manages the dynamic execution of the system and the SLAVE controllers are responsible for the management of the processes and FIFO channels. Thus, the MASTER controller distributes the actual computation to the SLAVE controllers. Once the SLAVE controllers performed their work, they go to sleep until the MASTER controller sends them a new job. This makes the proposed runtime manager very lightweight in the sense that it does not affect the execution of the process network.

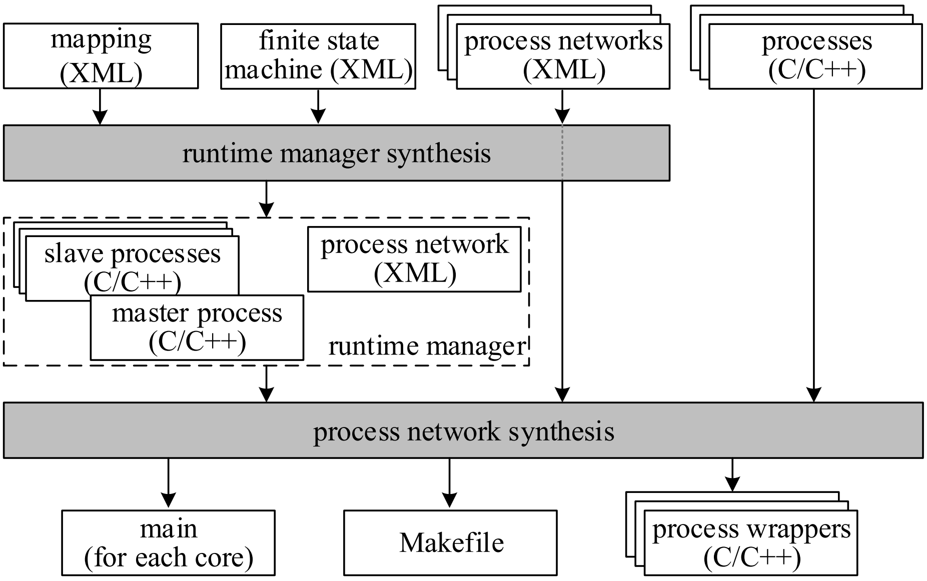


Fig. 2.11: Program synthesis flow to generate the source code of the runtime manager and the process wrappers.

Process Network Synthesis

Finally, the process network synthesis step embeds each process into a POSIX thread and creates a MAIN function for each core. Embedding each process into a POSIX thread is achieved by a process wrapper that repeatedly calls the FIRE procedure of the process. The process definition is then stored together with the process wrapper as a dynamic library in the file system so that it can be loaded on request of a SLAVE controller. The MAIN function has two tasks. First, it starts the LISTENER thread. Then, it initializes the processes and channels of the runtime manager and turns the control of the system over to the MASTER controller.

2.7 Evaluation

In this section, we provide evaluation results by means of a prototype implementation of DAL demonstrating the capabilities of the proposed scenario-based design flow. The goal is to answer the following questions. *a*) What are the limitations of the hierarchical control mechanism? *b*) What is the overhead of an optimized software stack generated by the proposed design flow? *c*) Does the proposed design flow provide enough flexibility to design complex and efficient embedded systems? To answer these questions, we evaluate the performance of synthetic and real-world benchmarks on a quad-core processor and on the SCC.

2.7.1 Experimental Setup

Intel Quad-Core Processor

The considered workstation has a quad-core Intel i7-2720QM processor clocked at 2.2 GHz (hyper-threading is deactivated) and 8 GBytes of memory. DAL runs on top of Linux and uses the POSIX library to execute multiple processes in parallel. Processes are stored as dynamic libraries, which are loaded and dynamically linked when the process is started. The used compiler is G++-4.5.3 with optimization level O2.

Intel SCC Processor

The prototype implementation of DAL presented in Section 2.6 has been used to execute KPNs on the Intel SCC processor [HDH+10]. The considered processor is configured to run the cores at 533 MHz and both the routers and the DDR3 RAM at 800 MHz. A Linux image with kernel 2.6.32 has been loaded onto each core and the RCKMPI library [URK11] has been configured to use the default channel implementation, i.e., the SCCMPB channel. In all experiments, the compiler is ICC 8.1 with optimization level O2.

2.7.2 Control Mechanism

To measure the overhead of the hierarchical control mechanism, multiple streaming applications have been executed concurrently on the Intel i7-2720QM processor. We configured DAL to consider the processor as an architecture with three communication layers whereby the inter-tile communication layer just consists of one tile with four cores. The workload is then split between two controllers. The MASTER controller is aware of the applications and the SLAVE controller is responsible for installing and removing processes and FIFO channels. As there is only one SLAVE controller in the system, basically, this one would be able to also process the tasks that are assigned to the MASTER controller. However, we selected this configuration as it enables us, on the one hand, to simulate the general case of having more than one SLAVE controller and, on the other hand, to measure separately the overhead generated by the behavioral dynamism and by the interactions with the operating system.

The application set consists of two single-process applications, namely the `fullload` application and the `pulse` application. The `fullload` application computes a predefined set of operations before stopping. Therefore, its execution time only depends on the other processes that are running on the same core. The `pulse` application sleeps for a certain time interval, the so-called switching time. Then it sends an event to the runtime manager that tells the controller to stop and restart the `pulse`

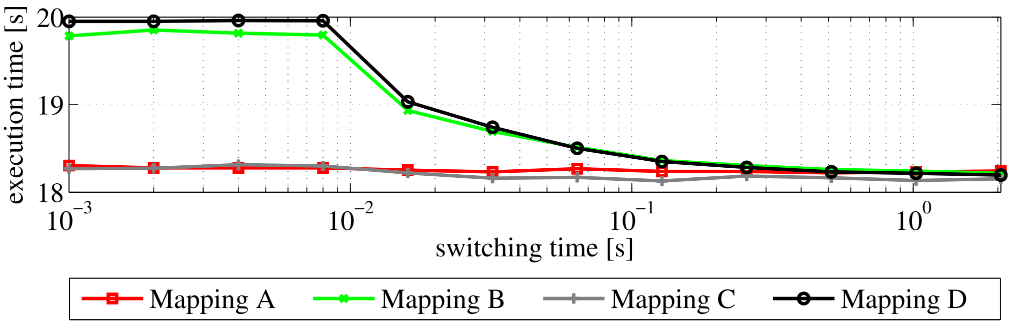


Fig. 2.12: Comparison of the time to execute the `fullload` application in different mapping configurations. The different mapping configurations are detailed in Table 2.2.

application. The overhead of the control mechanism is estimated by comparing the absolute execution times of the `fullload` application for different mappings, see Table 2.2 for the detailed mapping configurations.

Tab. 2.2: Mapping configurations to measure the overhead of the hierarchical control mechanism.

	core 0	**core** 1	**core** 2	**core** 3
Mapping A	MASTER	SLAVE	`fullload`	`pulse`
Mapping B	MASTER	SLAVE, `fullload`	-	`pulse`
Mapping C	MASTER, `fullload`	SLAVE	-	`pulse`
Mapping D	MASTER, SLAVE, `fullload`	-	-	`pulse`

In Fig. 2.12, the absolute time to execute the `fullload` application is compared for the four mapping configurations and different switching times. The switching time is the time interval between two scenario change requests. As the `fullload` application runs in all scenarios, it is not directly affected by the scenario changes and its absolute execution time depends only on the workload of the other processes that are running on the same core. Therefore, we can use the absolute execution time of the `fullload` application as an indicator for the overhead generated by the controllers.

While the MASTER controller generates no overhead, running the `fullload` application on the same core as the SLAVE controller increases the absolute execution time of the `fullload` application. If the SLAVE controller and the `fullload` application are running on the same core, the execution time of the `fullload` application increases by 1.3 % for a switching time of 64 ms and by 8.1 % for a switching time of 1 ms.

Overall, the results show that the interactions with the operating system are considerably more expensive than the management of the behavioral dynamism. This supports our decision to use a hierarchically organized runtime manager where multiple controllers are simultaneously interacting with the different operating systems.

2.7.3 System Specification and Optimization

Next, we show that the proposed design flow provides enough flexility to design complex embedded systems and we evaluate the performance of the proposed optimization strategy. For this purpose, we first design a multistage Picture-in-Picture (PiP) benchmark for embedded video processing systems. Afterwards, we compare the performance of different mapping strategies when mapping the PiP benchmark onto Intel's SCC processor.

Picture-in-Picture Benchmark

We extend the Eclipse SDK with an editor called DALipse[1] to design benchmarks for the proposed design flow. DALipse enables the system designer to visually specify finite state machines, process networks, and abstract models of many-core SoC platforms. Figure 2.13 shows a screenshot of DALipse with the finite state machine of the considered PiP benchmark. The benchmark is composed of eight scenarios and three different video decoder applications. The `HD`, `SD`, and `VCD` applications process high-definition, standard-definition, and low-resolution video data, respectively. The benchmark has two major execution modes, namely watching high-definition (scenario '*HD*') or standard-definition videos (scenario '*SD*'). In addition, the user can pause the video or watch a preview of another video by activating the PiP mode (i.e., starting the `VCD` application). Due to resource restrictions, the user is only able to activate the PiP mode when the `SD` application is running or paused, or the `HD` application is paused.

For illustration purpose, we use different implementations of a Motion JPEG (MJPEG) decoder as applications. The MJPEG standard [Wal92] is a video compression format in which each video frame is separately compressed as a JPEG image. The process network of the `VCD` application is depicted in Fig. 2.14 and the three different video decoder applications are summarized in Table 2.3. The MJPEG decoder is able to decode a certain number of frames in parallel. In particular, the 'ss' ("split stream") process reads the video stream from a playout buffer and dispatches single video frames to subsequent processes. The 'sf' ("split frame") process unpacks and predicts DCT coefficients so that the 'dec' ("decode") process can decode one DCT block per activation. Finally, the 'mf' ("merge frame")

---

[1] DALipse is also available online for download at `http://www.dal.ethz.ch`.

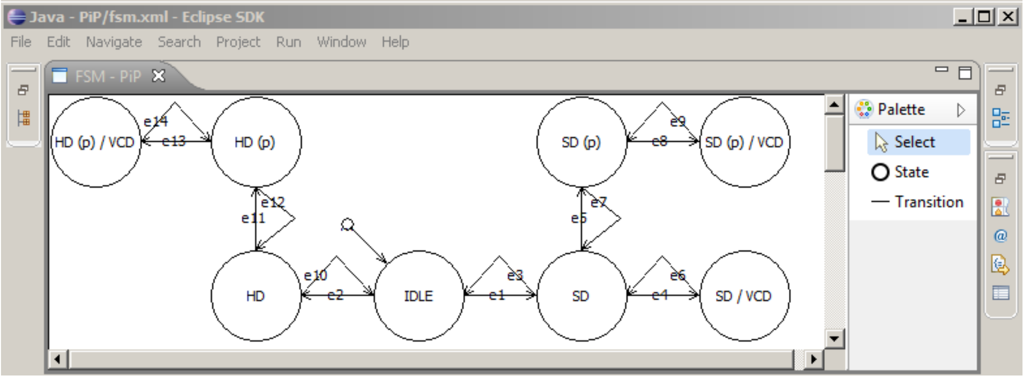


Fig. 2.13: Screenshot of DALipse with the finite state machine of the PiP benchmark considered in Section 2.7.3. Paused applications are indicated by a `(p)` following the name of the application. For instance, scenario '*HD(p)/VCD*' refers to the state where the application processing the high-definition video stream is paused and the application processing low-resolution video data is running.

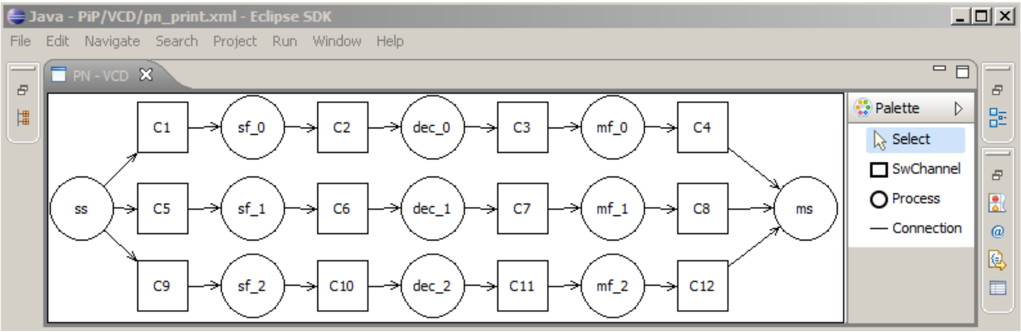


Fig. 2.14: Screenshot of DALipse with the process network of the `VCD` application.

process collects the DCT blocks, and the 'ms' ("merge stream") process collects the decoded frames. All three video decoders read their playout buffers at a constant rate of 25 frames/second. The maximum execution time of a process as well as the data volume per time unit and channel has been measured by executing the applications on Intel's SCC processor.

Mapping Optimization

Next, we study the performance of the hybrid mapping optimization strategy by comparing it with the performance of other mapping strategies.

For this purpose, we extend the PISA framework [BLTZ03] to solve the mapping optimization problem stated in Section 2.5.1. In particular, PISA is extended to calculate the collection M of optimal mappings so

Tab. 2.3: Configuration of the three considered video decoder applications of the PiP benchmark.

application	**resolution**	**pixels / frame**	**# processes**	**# channels**
HD	$1'280 \times 720$	921′600	98	128
SD	720×576	414′720	50	64
VCD	320×240	76′800	11	12

that exactly one mapping m of this collection is valid for each pair of application and scenario. Violations of bandwidth constraints are avoided by imposing a penalty on the maximum utilization. PISA solves the mapping optimization problem by either generating 1′000 random solutions and selecting the best of them as overall solution, or using the EA SPEA2 [ZLT01].

We compare the performance of four different mapping strategies when minimizing the maximum core utilization for different numbers of available cores on Intel's SCC processor:

- The DYNAMIC-OPTIMAL mapping strategy represents the hybrid design time and runtime mapping optimization strategy solved using the EA SPEA2.

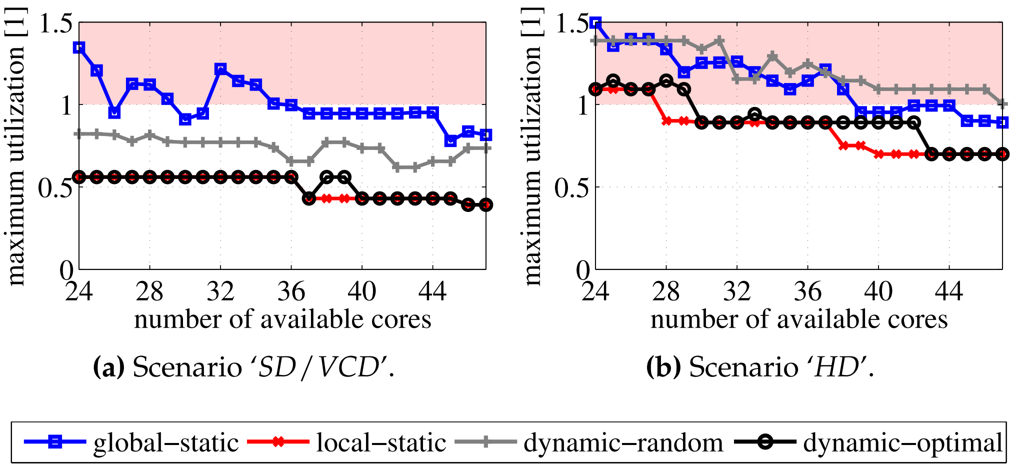
- The DYNAMIC-RANDOM mapping strategy represents the hybrid design time and runtime mapping optimization strategy solved by selecting the best of 1′000 random solutions.

- The GLOBAL-STATIC mapping strategy calculates a single static mapping for the system, i.e., it does not make use of the different execution scenarios. The GLOBAL-STATIC strategy is solved using the EA SPEA2.

- The LOCAL-STATIC mapping strategy calculates a single mapping for each scenario. Calculating a single mapping for each scenario individually might lead to the situation that an application has a different mapping in two connected scenarios. Therefore, the runtime-system must support process migration if the LOCAL-STATIC strategy is used. The EA SPEA2 is used to solve the LOCAL-STATIC strategy.

The results of this comparison are plotted in Fig. 2.15 for scenario '*SD / VCD*' and scenario '*HD*'. Utilizations larger than one imply that the mapping strategy is unable to find a valid mapping for the considered number of available cores.

As expected, the LOCAL-STATIC mapping strategy reduces the maximum utilization the most as it calculates the local optimal mapping for



(a) Scenario *'SD / VCD'*.

(b) Scenario *'HD'*.

Fig. 2.15: Comparison of the maximum core utilization for different numbers of available cores and different optimization strategies. The dynamic optimal mapping strategy represents the hybrid scenario-based mapping optimization strategy proposed in Section 2.5.1. Utilizations larger than one imply that the mapping strategy is unable to find a valid mapping for the considered number of available cores.

just one scenario. However, the unavoidable runtime support for process migration leads to non-negligible costs in terms of time and system overhead. The hybrid design time and runtime mapping optimization strategy, i.e., the dynamic-optimal mapping strategy, results in a utilization that is on average 0.01 (for scenario *'SD / VCD'*) and 0.05 (for scenario *'HD'*) larger than the utilization calculated by the local-static mapping strategy.

PISA is unable to find a valid mapping for the *'HD'* scenario when the global-static mapping strategy is used and less than 39 cores are available. However, PISA is able to find a valid mapping for 30 cores and the *'HD'* scenario if the dynamic-optimal mapping strategy is used. Compared to the global-static mapping strategy, the dynamic-optimal mapping strategy reduces the utilization on average by 0.51 for the *'SD / VCD'* scenario and 0.16 for the *'HD'* scenario.

As the dynamic-optimal mapping strategy does not only optimize a single scenario, its utilization may increase with number of available cores for certain execution scenarios. For example, the maximum utilization of execution scenario *'HD'* is slightly increased when moving from 32 to 33 available cores. Note that the selection of the solver has a high influence on the performance of the hybrid design time and runtime mapping optimization strategy. Selecting the best of 1'000 random solutions may result in a performance that is even worse than the global-static mapping strategy.

Overall, we have shown that the proposed design flow provides enough flexility to design complex embedded systems with reasonable effort. Furthermore, the proposed hybrid mapping strategy has been proven to be a scalable mapping strategy suited for the model-driven development of complex multi-functional embedded systems.

2.7.4 System Characterization

Next, we quantify the overhead of an optimized software stack that is generated by the DAL design flow. For this purpose, we execute synthetic and real-world applications that have been specified by the API proposed in Section 2.4, the on the Intel SCC processor.

Data Transfer Rate

A synthetic application consisting of two processes and one FIFO channel has been designed to measure the data transfer rate between two cores. The application executes 100'000 iterations and in one iteration, the source process writes one token to the FIFO channel and the sink process reads the token from the FIFO channel. No other processes except the runtime manager are running on the SCC. The mappings are selected so that the source process and the sink process are assigned to either cores on the same tile or to cores that are a certain number of hops[2] apart from each other. Figure 2.16 shows the data transfer rate between two cores whereby the token size is varied between 32 bytes and 16'384 bytes. The experiment has been repeated for two capacities of the FIFO channel and three different mappings.

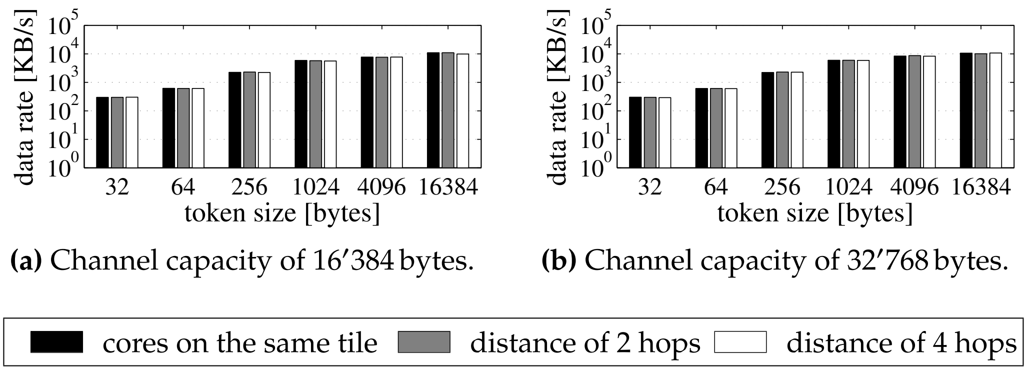
The observed peak data transfer rate is 11 Mbytes/s. While mapping and capacity have small influence on the data transfer rate, the data transfer rate significantly increases with the size of a single token. This might be as the used software stack is the bottleneck of the considered communication infrastructure.

Runtime-System Overhead

As the runtime manager is idle after performing some work, the overhead of the runtime-system can mainly be assigned to the LISTENER thread that periodically checks if new data is available. To measure this overhead, we use a sequential implementation of the MJPEG decoder. Decoding 5'000 frames takes about 162.0 s if the MJPEG decoder is executed in parallel to the LISTENER thread and 158.9 s if it is executed as an individual

---

[2] A hop represents the path between two neighboring tiles.



(a) Channel capacity of 16′384 bytes. **(b)** Channel capacity of 32′768 bytes.

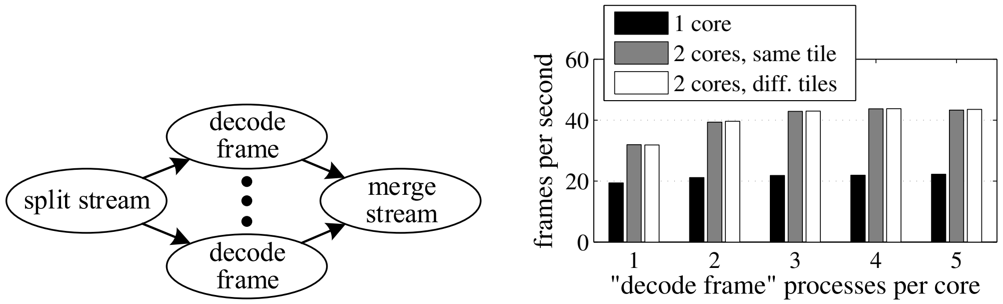
Fig. 2.16: Data transfer rate between two cores on Intel's SCC processor for three different mappings.

application. Thus, the measured overhead of the considered software stack is less than two percent.

Context Switching Overhead

To characterize the effect of multi-processing on a single core, we consider a distributed implementation of the MJPEG decoder, see Fig. 2.17a for the process network. The network has multiple 'decode frame' processes and each process decodes one frame per iteration. In Fig. 2.17b, the decoded frames per second using the MJPEG algorithm are compared for implementations mapping a different number of 'decode frame' processes onto one core. Furthermore, the graph differs between three configurations. First, only one core, then both cores of a tile, and finally two cores of different tiles are used to execute the 'decode frame' processes. It shows that the frame rate significantly increases if two processes are mapped onto the same core as communication and computation can partially overlap. This is also illustrated in Fig. 2.10. While one process is waiting until the write operation is completed, other processes can execute on the same core. Furthermore, the frame rate does not decrease even if five processes are mapped onto each core indicating that the MJPEG decoder application has a low multi-processing overhead.

Overall, the overhead introduced by the software stack for executing DAL benchmarks on Intel's SCC processor is reasonable small. In fact, the application-independent parts, namely the implementation of FIFO channel communication and the synchronization of the processes running on the same core, can be implemented once and reused in any application so that the optimization effort must be conducted only once for all applications.



(a) KPN of the considered distributed implementation of the MJPEG decoder.

(b) Decoded frames per second for different numbers of 'decode frame' processes on one core.

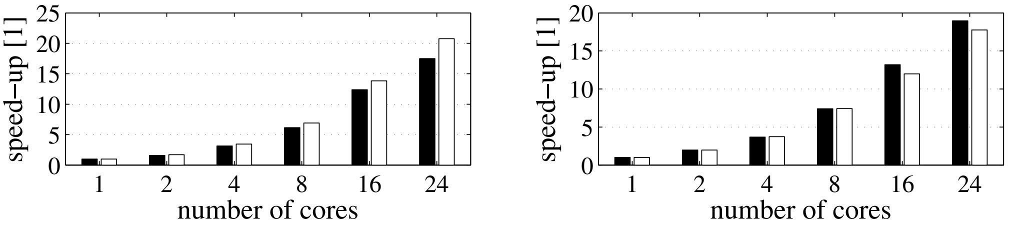
Fig. 2.17: Evaluation of the context switching overhead using a distributed implementation of the MJPEG decoder.

2.7.5 Speed-up due to Parallelism

Finally, we evaluate the speed-up due to the available number of cores for three different applications. Besides the previously introduced distributed implementation of the MJPEG decoder, a ray-tracing algorithm, and an MPEG-2 decoder are studied. The ray-tracing algorithm generates an image of 100 × 100 pixels and can analyze multiple rays concurrently. We map either one or two such processes onto one core. The MPEG-2 decoder decodes multiple macroblocks concurrently. We again map either one or two such processes onto one core.

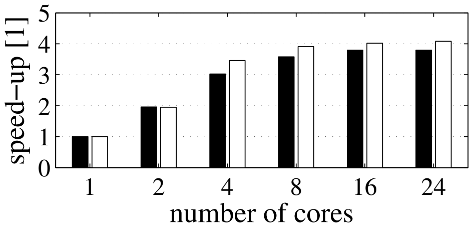
In Fig. 2.18, the speed-up is compared for implementations running on a different number of cores. The speed-up is calculated with respect to an implementation running on a single core. The maximum speed-up that can be achieved is 20.7 for the MJPEG decoder application. As MJPEG is an intraframe-only compression scheme, the frames can be decoded in parallel on different cores. The ray-tracing algorithm achieves a speed-up of almost 20 on 24 cores. As each ray can be analyzed individually, the ray-tracing algorithm is well-suited for parallelization. The speed-ups achieved with the MPEG-2 application are much smaller than with the previous applications. Due to data-dependencies between the frames, the MPEG-2 application only achieves a speed-up of about 4.1. The speed-up increases linearly for a small number of cores before collecting and distributing frames become the bottleneck for higher parallelization.

Overall, the results demonstrate that the proposed design flow enables system designers to efficiently exploit the hardware parallelism of many-core platforms. The previous evaluation has also shown that the maximum achievable performance depends on the application structure. On the one hand, if the number of parallel processes is selected too small, not all cores can be utilized. On the other hand, the results observed for



(a) MJPEG decoder.

(b) Ray-tracing algorithm.



(c) MPEG-2 decoder.

Fig. 2.18: Speed-ups of three benchmarks for a varying number of cores on Intel's SCC processor.

the ray-tracing algorithm show that mapping too many processes onto one core can result in inefficient implementations. The Expandable Process Network (EPN) semantics, which will be the topic of the next chapter, will tackle this issue by automatically selecting the best degree of task, data, and pipeline parallelism.

2.8 Summary

Modern embedded systems are becoming multi-functional by featuring multiple applications that are running simultaneously in different combinations at different moments in time. Only if the system is able to react to runtime variations by reconfiguring the mapping of the applications onto the architecture, the computing power offered by future many-core SoC platforms can be exploited efficiently. Moreover, with each application having its own performance constraints, it must be guaranteed that each application meets its constraints independently of the other applications.

In this chapter, it has been shown that these goals can be met simultaneously using hybrid design time and runtime mapping strategies. For this purpose, a scenario-based design flow (the DAL design flow) for the model-driven development of heterogeneous embedded many-core SoCs has been proposed. The DAL design flow supports the design, the optimization, and the simultaneous execution of multiple streaming applications. Its input is a set of applications, each specified as a KPN, and a finite state machine specifying the interactions between the applications. Each state of the finite state machine represents an execution scenario, i.e., a certain use case of the system with a predefined set of running or paused applications. The proposed hybrid design time and runtime mapping optimization strategy consists of two components. At design time, each application is assigned a set of optimized mappings and each mapping is individually valid for a subset of execution scenarios. At runtime, hierarchically organized controllers monitor the system and use the pre-calculated mappings to start and stop applications according to the finite state machine. To include the evaluation of all possible failure scenarios in the design time analysis, spare cores and tiles are allocated during design time optimization and used by the runtime manager as target for process migration. As no additional design time analysis is required, the proposed approach leads to a high responsiveness to faults.

Based on a prototype implementation of DAL targeting Intel's SCC processor, we have demonstrated that complex multi-functional embedded systems can be designed in DAL with reasonable effort. DAL is freely available for download at `http://www.dal.ethz.ch` and has been successfully applied in several academic projects both as front-end and back-end. For instance, in the context of the EU FP 7 project EURETILE, DAL is used as front-end to design and optimize many-tile systems with up to 200 tiles. On the other hand, the Orcc compiler [SWNR10] has been extended to use the DAL design flow as back-end [Bou14] to execute applications written in the RVC-CAL actor language [MAR10] on heterogeneous many-core systems.

3

Expandable Process Networks

3.1 Introduction

As discussed in the previous chapter, the functionality of a modern embedded system can change over time. In this thesis, we model such a multi-functional behavior by a set of execution scenarios. Each execution scenario represents a different set of running or paused applications and a runtime manager is in charge of managing the computing power as well as allocating resources to each application.

In order to exploit the available resources efficiently, the amount of computing power available to a single application may vary between execution scenarios. In particular, we assume that the number of processing cores available to an application can change over time. For instance, the set of computing resources available to a certain application may contain many slow processing cores in one execution scenario and just a few fast processing cores in another execution scenario. Even though such a dynamic resource allocation leads to a high resource utilization, the drawback is that the system designer does not know the available computing resources anymore when specifying and programming the application.

This challenge can be tackled by considering the parallelism provided by the application. However, if the application is specified statically, the maximum number of cores that an application can utilize simultaneously is limited to the number of processes. On the other hand, having too many parallel processes might result in inefficient implementations of the application due to overheads in scheduling and inter-process communication. To overcome these limitations, the application's degree of parallelism must be adapted so that it matches with the available resources. In fact, previous work has already shown that large performance gains in terms of throughput [CLS+12, SLA12, ZBS13] or energy consumption [BL13] can be obtained if the application's degree of parallelism is refined before the execution. However, these approaches are limited to either programming models with statically specified data production and consumption rates or the replication of stateless processes.

Overview

In this chapter, we argue that for a certain class of applications, namely applications that are specified as process networks, the application can be specified in a manner that enables the automatic exploration of task, data, and pipeline parallelism [GTA06, YH09]. Task parallelism is achieved by executing different processes on different cores. In contrast, data parallelism refers to creating multiple instances of a process that perform the same task on different pieces of distributed data. Finally, pipeline parallelism is achieved by splitting a process into stages and assigning each stage to a different processing core.

We call the proposed model of computation Expandable Process Network (EPN). The EPNs model of computation extends conventional streaming programming models by abstracting several possible granularities in a single specification. To this end, it specifies an application as a top-level process network that can be refined by hierarchically replacing individual processes. This enables the automatic exploration of task, data, and pipeline parallelism by two refinement strategies, namely *replication* and *unfolding*. Replicating processes increases data parallelism and structural unfolding of a process increases the task and pipeline parallelism by hierarchically instantiating more processes in the process network. Furthermore, as recursive algorithms are commonly used in mathematical [AOB93] and multimedia [BSP06] applications, we study the recursive description of processes as a structural unfolding method.

For illustration purpose, we apply the proposed semantics of EPNs to Kahn Process Networks (KPNs) [Kah74]. We will show that the EPN specification can be used to synthesize multiple instances of the same application, each optimized for a different execution scenario or different resource availabilities, automatically. In order to react to changes in the available resources, we propose a novel technique that transparently transforms the application from one instance into an alternative instance without discarding its program state. The results of the conducted evaluations on two many-core platforms show that having an abstract application specification clearly outperforms a static specification when the available computing resources are not known at design time.

Outline

The remainder of the chapter is organized as follows: We review related work in the next section. In Section 3.3, the proposed concepts are integrated into the Distributed Application Layer (DAL) design flow, which has been described in the previous chapter. In Section 3.4, the proposed semantics of EPNs is described formally and the concepts of replication and unfolding are detailed. In Section 3.5, the optimization problem for parallelizing and mapping applications specified as EPNs is formulated and solved using a novel optimization heuristic. In Section 3.6, a technique to transform an application transparently into an alternative process network is described. Finally, the results of the performed case studies are presented in Section 3.7.

3.2 Related Work

In this section, we review recent efforts to adapt the degree of parallelism of a streaming application. First, we summarize the basic ideas of various transformation techniques proposed to refine the degree of parallelism of a streaming application. Afterwards, we review efforts to achieve dynamic load balancing at runtime.

Techniques to Refine an Application at Design Time

In the last few years, various models of computation for specifying signal processing and streaming multimedia applications have been proposed, which have different characteristics in terms of decidability and expressiveness, see, e.g., [Bam14] for an overview. Applications specified as KPN [Kah74], for instance, are determinate, provide asynchronous execution, and are capable to describe data-dependent behavior. The Synchronous Dataflow (SDF) model of computation [LM87] restricts each process to produce and consume a fixed number of tokens in every iteration so that the application is amenable to design time scheduling techniques. As the SDF model has limited expressiveness, several extensions have been proposed that offer the ability to specify flexible and dynamic behavior and still preserve the capability to analyze the application statically. For instance, the Synchronous Piggybacked Dataflow (SPDF) model of computation [PJH02] enables $if - else$ and for clauses in SDFs. Hierarchical representations of process networks are allowed in some design frameworks, e.g., [BLL+05], but they are semantically equivalent to basic process networks, as they can be flattened at design time. However, none of these extensions bring any benefit in terms of parallelism, as they keep the process network topology unchanged. Dynamic representations such as Reactive Process Networks (RPNs) [GB04] offer the possibility to capture runtime topology changes by deactivating and activating independent parts of the process network. Even though the topology of active process networks is no longer static, the degree of parallelism is still statically determined at specification time.

Formal design and program transformations are considered to be an efficient approach to optimize an application towards the final architecture. A survey of existing transformation methods for functional programs is given, e.g., in [PP96]. The previously discussed models of computation have in common that they specify the application in a high abstraction level suitable for design transformation. However, due to simplicity, most design transformation strategies focus on applications modeled as SDF graphs. In [ML94], successive iterations of an SDF graph are considered as a block enabling the concurrent execution of multiple instances of a single process. Our approach, in contrast, uses replication to execute multiple instances of a single process concurrently. In [PL95], a clustering technique for SDF graphs is proposed that unfolds the graph completely, and then uses clustering techniques to reduce the number of processes per processing core to optimize scheduling. On the other hand, our technique only unfolds a graph if a performance gain is achieved by the additional parallelism. Various transformation rules to balance a network are presented in the context of the ForSyDe methodology [SJ04], a synchronous computational model. In contrast, our technique is based on KPNs and achieves a finer granularity by applying the proposed semantics to individual processes.

More design transformation and refinement strategies are presented, e.g., in [GTA06, SLA12, ZBS13]. They use the concept of fusion and fission operators to change the number of replicas of stateless processes at the mapping stage. In particular, *fusion* is used to coarsen the SDF granularity to increase the computation to communication ratio and *fission* is used to distribute data parallel tasks to multiple cores. However, in contrast to our approach, structural expansions have not been considered in these works. Therefore, the maximum degree of task and pipeline parallelism is still upper bounded by the number of processes.

Another difference to the above described techniques is that our approach enables the specification of recursive behavior as a structural unfolding method. Recursion is a procedure that repeatedly calls itself, and is typically used in programming to divide a problem into multiple subproblems with the same repetitive behavior. In case of a huge amount of independent data that needs to be processed in a similar manner, recursive implementations are of practical use due to their simplicity.

There is a wide range of mathematical algorithms that can be implemented recursively [CLR+01] and even multimedia applications, such as ray-tracing [BSP06], can be specified as recursive algorithms.

In summary, conventional process networks or dataflow models are too static and monolithic to explore different application structures at design time. Moreover, they cannot represent recursive dependencies, which are necessary to describe certain classes of applications effectively. The proposed EPN semantics extends conventional models of computation to specify streaming applications. It enables the exploration of an efficient application structure by exploiting task, data, and pipeline parallelism.

Techniques to Adapt an Application at Runtime

When the available resources may change at runtime, programming models and techniques are required that can adapt the application's degree of parallelism. Flextream [HCK+09], for instance, is a flexible compilation framework to adapt the mapping of a streaming application dynamically. It refines the process network at design time by replicating stateless processes and using the largest possible resource allocation as target platform. At runtime, it assigns the processes that have been assigned to cores that are not available originally, to the remaining cores. However, the memory usage of the application is virtually independent of the available cores and the application might have a considerable scheduling overhead on a single core. In contrast, our work proposes to synthesize multiple instances of the same application automatically and each instance is optimized for a different number of available cores. In order to transform the application from one instance into an alternative instance, we propose a transformation technique that adapts the application's degree of parallelism without discarding its program state. Runtime task duplication is used in [CLS+12] to maximize the application's throughput. The technique replaces stateless processes by a master thread that distributes the actual work among its sibling threads. When the available processing cores are changed, the number of sibling threads is increased or decreased to improve the throughput. In contrast, our work proposes structural unfolding of processes by process networks as a mechanism to also refine stateful processes.

A dynamic scheduling approach for streaming applications specified as SDF graphs is proposed in [LCC12]. The approach uses the fusion and fission operators to generate a schedule that maximizes the throughput of the application for the available amount of resources. StreaMorph [BL13] is a technique to adapt SDF graphs at runtime by performing a reverse sequence of executions to bring the graph into a known state. In contrast to the previous two approaches, our technique does not assume a static schedule and is therefore applicable to more complex models of computation than SDF graphs. Furthermore, our approach supports stateful processes, a key characteristic of general process networks that is not supported by the previously discussed techniques.

Finally, a different approach to achieve dynamic load balancing, the overall goal of the proposed approach, is the concept of task stealing [BL99]. Even though efficient implementations for shared-memory systems have been presented (e.g., [LPCZN13]), task stealing approaches still suffer from communication overheads, in particular on distributed memory systems [LHCZ13]. However, this overhead can be reduced if tasks are assigned to specific processing cores. In addition, task stealing provides only limited options to exploit pipeline parallelism. In contrast, our technique uses pipeline parallelism to split large (stateful) tasks into sub-tasks.

3.3 System Design

To include the proposed concepts in the system design, we extend the DAL design flow shown in Fig. 1.3 with an additional step called *design transformation*. The part of the design flow that is responsible to automatically synthesize multiple instances of an application is detailed in Fig. 3.1. The concepts proposed in this chapter will be integrated into this design flow.

The inputs to the design flow shown in Fig. 3.1 are an application specified as an EPN and an abstract specification of the target architecture that follows the specification proposed in Section 2.3. In a first step, possible sub-architectures are identified. A sub-architecture contains only a subset of all cores of the target architecture. It is used to represent the fact that the runtime manager might not assign all available processing cores to an application, but only a subset of them. Particularly with regard to the DAL design flow, one sub-architecture is allocated per execution scenario that contains the application. In case of no predefined execution scenarios, the set of sub-architectures generated for a given architecture should cover the variety of processing core subsets that the runtime manager can assign to an application. For a homogeneous platform, such as the Intel Single-chip Cloud Computer (SCC) processor, each sub-architecture might only differ in the number of processing cores. In case that the target architecture is heterogeneous, the number of possible sub-architectures might be larger. However, the number can be reduced by considering the symmetry of the architecture or by selecting only a subset of all possible sub-architectures and ignoring sub-architectures with similar computing

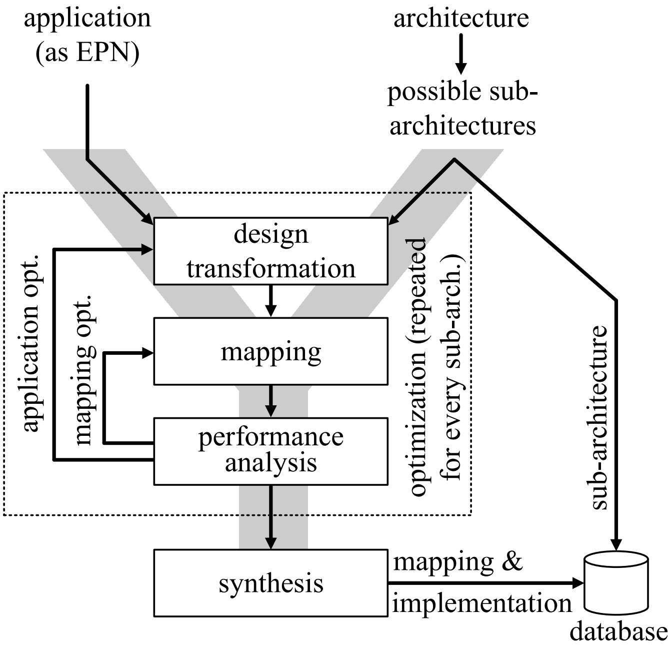


Fig. 3.1: Part of the modified DAL design flow to synthesize multiple instances of an EPN application automatically. Each instance has a different degree of parallelism and is optimized for a different resource availability.

power. Even more, the set of valid sub-architectures might be reduced if the application has specific performance requirements that can only be met if the sub-architecture provides a minimum amount of computing power.

The structure and the mapping of the application are optimized separately for each sub-architecture, aiming to maximize the throughput of the application. Note that the concepts proposed in this chapter are not restricted to the maximization of the throughput, but can also be applied to optimize other performance metrics such as the energy consumption. The optimization stage consists of the design transformation, the mapping, and the performance analysis. In the design transformation step, replication and unfolding are used to explore the parallelism. We will detail these refinement strategies later in Section 3.4. Afterwards, a mapping is calculated and the throughput of the application is evaluated. The design transformation and mapping optimization steps are repeated until a degree of parallelism is found that fulfills the performance requirements of the system. Based on the information obtained in the optimization steps, a concrete (replicated and unfolded) process network is generated during synthesis. Finally, the description of the sub-architecture is stored in a database together with the synthesized process network and the respective mapping information.

The runtime manager uses the database to select an optimized implementation based on the available computing resources. In case the computing resources available to a certain application change at runtime, e.g., as other applications are started or stopped, the runtime manager must transform the application into an alternative process network without discarding the program state of the application.

3.4 The Semantics of Expandable Process Networks

EPNs extend conventional streaming programming models in the sense that several possible granularities are abstracted in a single specification. In other words, an application specified as an EPN has a top-level process network defining the initial network. The initial network can be refined by replicating processes or by replacing processes by other process networks. We call the first refinement strategy *replication* and the second one *unfolding*.

In this section, we formally specify the application model. First, we discuss the semantics of EPNs. Then, we describe the concepts of replication and unfolding. Finally, we propose a high-level API for EPNs.

3.4.1 Application Specification

The proposed semantics of EPNs is applied to the KPN [Kah74] model of computation that has been discussed in Section 2.4. To repeat, a process network is a tuple $p = \langle V, Q \rangle$ with the set of processes V and the set of FIFO channels $Q \subseteq V \times V$, see Definition 2.3. However, as the functionality of an EPN process may not only be specified in a high-level programming language, but also as another process network, we extend the definition of a process, as follows.

Def. 3.1: **(EPN Process)** *An EPN process* v *is a tuple* $v = \langle name, type, replicable, in, out \rangle$, *where name is a unique string identifier, type* $\in \{behav, struct\}$ *describes the type of the instantiated process, replicable* $\in \{false, true\}$ *indicates if a process can be replicated, in* $\subseteq Q$ *denotes the set of incoming channels of process* v, *and out* $\subseteq Q$ *denotes outgoing channels.*

Except the elements *type* and *replicable*, the proposed process description is identical to the specification of a KPN process as described in Definition 2.3. The *type* identifier reveals that some processes do not only have a behavioral, but also a structural description. The type *struct* specifies that the process has a behavioral and a structural description, while the type *behav* specifies that the process has only a behavioral description.

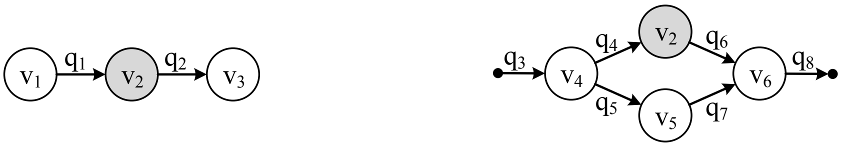
The behavioral description of a process v specifies the functionality of process v in the DAL programming language described in Section 2.4. The structural description of a process v defines the functionality of process v as another process network, i.e., as a set of processes and channels. Both the behavioral and structural descriptions have to be functionally equivalent in the sense that, for a given sequence of input tokens, they produce the same sequence of output tokens.

An EPN abstracts several possible granularities in a single specification. More detailed, an EPN is specified as a top-level process network and the processes of the top-level process network can be replicated or replaced by other process networks (so-called refinement networks).

Def. 3.2: **(Expandable Process Network)** *An Expandable Process Network (EPN) is a tuple* $e = \langle P, u, l, p_{org} \rangle$*, where* P *is a set of process networks,* u *and* l *are transformation functions, and* p_{org} *is the top-level process network from which processes may be further replicated or structurally unfolded.*

The top-level process network is the most coarsened process network abstraction of the application; it might even consist of only one process. In order to specify functions u and l, we define the set of all processes of EPN e as $V^e = \bigcup_{p=\langle V,Q \rangle \in P} V$ and the set of all channels of EPN e as $Q^e = \bigcup_{p=\langle V,Q \rangle \in P} Q$. Function $u : V^e \rightarrow P$ maps a process v to a corresponding refinement network $p = u(v)$. In other words, $u(v)$ represents the structural specification of process v. Function $l : Q^e \rightarrow Q^e$ maps a channel q to a corresponding channel $l(q)$ representing the match between the input and output channels of process v and the input and output channels of the structural specification $u(v)$ of v.

Ex. 3.1: *Consider the example specification shown in Fig. 3.2. The EPN* $e = \langle \{p_{org}, p_{v_2}\}, l, u, p_{org} \rangle$ *has the top-level process network* $p_{org} = \langle \{v_1, v_2, v_3\}, \{q_1, q_2\} \rangle$. $v_1 = \langle `v_1', behav, false, \emptyset, \{q_1\} \rangle$ *and* $v_3 = \langle `v_3', behav, false, \{q_2\}, \emptyset \rangle$ *are ordinary processes which have no further unfolding capabilities.* $v_2 = \langle `v_2', struct, true, \{q_1\}, \{q_2\} \rangle$ *is a process of type* struct*, which has both a behavioral and a structural description. The structural description of* v_2 *is the refinement network* $p_{v_2} = \langle \{v_2, v_4, v_5, v_6\}, \{q_3, q_4, q_5, q_6, q_7, q_8\} \rangle$*. Note that process* v_2 *appears in both* p_{org} *and* p_{v_2} *enabling recursive unfolding. As* v_2 *is the only process of type* struct*,* u *returns null for all inputs except* v_2 *and* $u(v_2) = p_{v_2}$*. Similarly, function* l *returns null for all inputs except the input and output channels of* v_2*. However, as* v_2 *occurs in both* p_{org} *and* p_{v_2}*, function* l *contains four assignments, namely* $l(q_1) = q_3$*,* $l(q_2) = q_8$*,* $l(q_4) = q_3$*, and* $l(q_6) = q_8$*.*



(a) Top-level process network p_{org}. **(b)** Structural description of process v_2.

Fig. 3.2: Exemplified specification of an EPN $e = \langle\{p_{org}, p_{v_2}\}, l, u, p_{org}\rangle$.

3.4.2 Refinement Strategies

EPNs enable an efficient, architecture independent specification of process networks. In fact, the top-level process network p_{org} of an EPN $e = \langle P, u, l, p_{org}\rangle$ can be refined by applying the replication and unfolding concepts. Each process refinement results in a new process network $p_a = \langle V_a^e, Q_a^e\rangle$ that has the same functionality as the top-level network. In the following, we will detail the concepts of replication and unfolding.

Replication

Handling parallelism inside a process is typically difficult because a process is mapped as a whole onto a single processing core. Artificially parallelizing the process using conventional parallel processing APIs (e.g., MPI [Pac96] or OpenMP [CJVDP07]) is undesirable as the implicit parallelism makes design time analysis impossible. Exposing this information at the process network level is more beneficial as it results in higher predictability and better mapping decisions.

In the EPN semantics, the step of handling parallelism inside a process is called *replication*. Replication is particularly applicable to algorithms that have a high data level parallelism, as it is often the case with algorithms optimized for SIMD processors. Typical examples are deinterlacing algorithms used to convert interlaced videos, image noise reduction algorithms, or video decompression and compression algorithms. Consider, for example, a video decoder that uses intraframe-compression. As there is no relation between the frames, multiple frames might be processed in parallel on different cores. Furthermore, various video compression algorithms support the segmentation of a frame into macroblocks and these macroblocks can be processed simultaneously.

The concept of replication has already been widely used to improve the performance of process networks (e.g., [GTA06, TBHH07]). Typically, the bottleneck process has been replicated to improve the overall performance. However, in contrast to all these concepts, we do not propose to define the number of replicas statically at specification level, but we argue

Algorithm 3.1 Replicate process $v = \langle name, type, replicable, in, out\rangle \in V$ of process network $p = \langle V, Q\rangle$ χ times.

```
$V \leftarrow V - \{v\}$                                   ▹remove process v

for $i = 1 \rightarrow \chi$ do                            ▹generate a replicated process
    $v_i \leftarrow \langle name^{\{i\}}, type, false, \emptyset, \emptyset\rangle$
    $V \leftarrow V \cup \{v_i\}$
end for
                                                           ▹for each incoming channel
for all $q = \langle v_{src}, v_{dst}\rangle \in Q$ s.t. $v_{dst} = v$ and $v_{src} <> v$ do
    $Q \leftarrow Q - \{q\}$                               ▹remove channel q
    for $i = 1 \rightarrow \chi$ do
        $q_i \leftarrow \langle v_{src}, v_i\rangle$       ▹add a replicated channel
        $Q \leftarrow Q \cup \{q_i\}$
        $in$ of $v_i \leftarrow in$ of $v_i \cup \{q_i\}$
    end for
end for
                                                           ▹for each outgoing channel
for all $q = \langle v_{src}, v_{dst}\rangle \in Q$ s.t. $v_{src} = v$ and $v_{dst} <> v$ do
    replace $q$ with the set of replicated channels
end for
                                                           ▹for each self-loop channel
for all $q = \langle v_{src}, v_{dst}\rangle \in Q$ s.t. $v_{src} = v$ and $v_{dst} = v$ do
    $Q \leftarrow Q - \{q\}$                               ▹remove channel q
    for $i = 1 \rightarrow \chi$ do
        $q_i \leftarrow \langle v_i, v_{\{(i+1) \bmod \chi\}}\rangle$    ▹circular connection
        $Q \leftarrow Q \cup \{q_i\}$
        $out$ of $v_i \leftarrow out$ of $v_i \cup \{q_i\}$
        $in$ of $v_{\{(i+1) \bmod \chi\}} \leftarrow in$ of $v_{\{(i+1) \bmod \chi\}} \cup \{q_i\}$
    end for
end for
```

that it is the task of the optimizer to find a good degree of parallelism that maximizes the performance.

Algorithm 3.1 illustrates the steps to modify the topology of process network $p = \langle V, Q\rangle$ so that process $v = \langle name, type, replicable, in, out\rangle$ is χ times replicated. First, it removes process v and adds the replicated clones $v_i = \langle name^{\{i\}}, type, false, \emptyset, \emptyset\rangle$ with $i \in \{1 \ldots \chi\}$. Then, each incoming channel $q = \langle v_{src}, v\rangle$ of v is replaced by a set of replicated channels with $q_i = \langle v_{src}, v_i\rangle$. Similarly, each outgoing channel of v is replaced by a set of replicated channels. Self-loop channels, i.e., channels that connect v with itself, are handled last. For each self-loop channel, a new chain

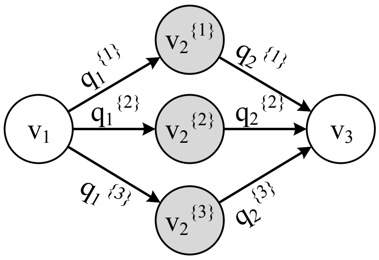


Fig. 3.3: The process network shown in Fig. 3.2a after replicating process v_2 three times.

of channels is introduced with one channel connecting v_i with v_{i+1}. As such a chain of channels introduces a circular dependency between the processes, it typically limits the maximum speed-up that can be obtained by replication. For instance, no speed-up can be achieved if v is reading from the self-loop channel at the beginning of the FIRE procedure and writing to it at the end of the FIRE procedure. In all other situations, the replicated copies of the process can still partly overlap their execution so that the system will achieve a speed-up higher than one.

Ex. 3.2: *Consider the EPN shown in Fig. 3.2. By replicating process v_2, it will be replaced by multiple instances of v_2. Figure 3.3 illustrates the process network if process v_2 has been replicated three times.*

Replicating processes with an internal state is supported by adding a self-loop channel representing the state of the process. Replicating two consecutive processes in a row is not allowed in order to prohibit complex communication behavior. If consecutive replication is needed for optimized performance, consecutive processes should be specified as a structural description, and then replicated altogether.

Unfolding

The EPN specification abstracts several possible granularities in a single specification. The step of exploring different degrees of task and pipeline parallelism by hierarchically instantiating more processes is called *unfolding* and is explained next.

Given an application specified as an EPN, a process of type *struct* can be unfolded by exposing internal parallelism at process network level. In other words, unfolding merely replaces the behavioral description of process v with its structural description $u(v)$. In addition, unfolding enables recursion as the structural representation $u(v)$ of process v may have process v as an internal process. In contrast to all previously proposed models, the maximum number of tasks is not statically determined. General instructions to unfold a process $v \in V$ of process network $p = \langle V, C \rangle$

Algorithm 3.2 Unfold process $v \in V$ of process network $p = \langle V, Q \rangle$ with refinement network $p_v = \langle V_v, Q_v \rangle$.

```
for all $v_i \in V_v$ do                                  ▷prefix $v$ to all names of $v_i \in V_v$
    $v_i.name \leftarrow v.name + v_i.name$
end for

for all $q_i = \langle src_i, dst_i \rangle \in Q_v$ do               ▷prefix $v$ to all names of $q_i \in Q_v$
    $src_i \leftarrow v.name + src_i$
    $dst_i \leftarrow v.name + dst_i$
end for

$V \leftarrow (V - \{v\}) \cup V_v$                          ▷remove $v$, add unfolded processes
$Q \leftarrow Q \cup Q_v$                                    ▷add unfolded channels

for all $q = \langle src, dst \rangle$ s.t. $dst = v$ do            ▷for each incoming channel
    for all $q_i = \langle src_i, dst_i \rangle \in Q_v$ do
        if $l(q) = q_i$ then                                 ▷find a match
            $src_i \leftarrow q.src$                          ▷adjust src of $q_i$
            $Q = Q - \{q\}$                                  ▷remove the unused channel
            break
        end if
    end for
end for

for all $q = \langle src, dst \rangle$ s.t. $src = v$ do            ▷for each outgoing channel
    remove all outgoing channels of $v$
end for
```

with refinement network p_v are given in Algorithm 3.2. First, the algorithm prefixes the name of process v to all unfolded processes and channels to keep them unique after design transformation. Afterwards, it removes process v and adds network p_v. Finally, it replaces each incoming and outgoing channel of v by the corresponding match in Q_v.

Ex. 3.3: *Consider again the EPN shown in Fig. 3.2. Process v_2 is of type* struct, *which means that it has a behavioral and a structural description. By recursively unfolding v_2, it will be replaced several times by refinement network p_{v_2}. Figure 3.4 illustrates the process network if process v_2 has been unfolded twice.*

Typically, the input of a process network limits the number of times that a process can be instantiated recursively. For instance, if recursively unfolding means that an array is split into two smaller arrays, the maximum recursion depth is defined by the length of the input array. In order

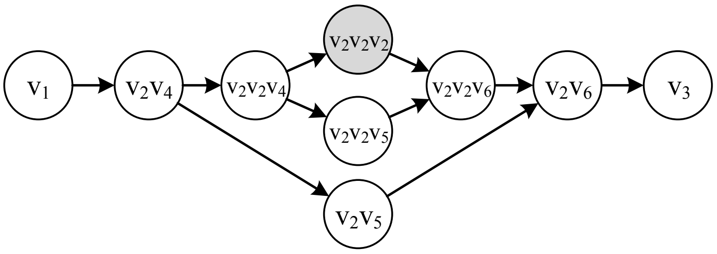


Fig. 3.4: A transformed process network of the EPN specified in Fig. 3.2. In particular, process v_2 has been unfolded twice.

to avoid deadlocks, the system designer has to either specify a termination condition for recursive unfolding (e.g., by knowing the minimum length of the array in the previous example) or guarantee that the application is not blocked if the input prohibits further recursion. The latter might be achieved by forwarding either the result or a predefined 'empty' string. In case the maximum recursion depth is known, the unfolding method can use this information to avoid blocking.

In summary, starting with the top-level process network, which is the most coarsened process network abstraction, all possible abstractions can be explored by applying the concepts of replication and unfolding.

Correctness

Next, we will show that the proposed refinement strategies preserve the correctness of the underlying model of computation if the conditions that are described in the following, are fulfilled.

Formally, the proposed refinement strategies transform the top-level process network p_{org} of EPN $e = \langle P, u, l, p_{org} \rangle$ into KPN $p_a = \langle V_a^e, Q_a^e \rangle$ with V_a^e the set of processes of p_a and Q_a^e the corresponding set of channels. A refinement strategy preserves correctness if, for a given input sequence, the refined process network p_a produces the same output sequence as the top-level process network p_{org}. Clearly, a first requirement for correctness is that each refinement network $p \in P$ is a valid KPN in the sense that it does not cause deadlocks.

Suppose now that the top-level process network $p_{org} = \langle \{v_1, v_2, v_3\}, \{q_1, q_2\} \rangle$ shown in Fig. 3.2a is given. To show that replication preserves correctness, process $v_2 = \langle v_2, behav, true, \{q_1\}, \{q_2\} \rangle$ is replicated χ times. Thus, channel q_1 is replaced by a set of channels $\{q_1^{\{1\}}, \ldots, q_1^{\{\chi\}}\}$, and channel q_2 is replaced by a set of channels $\{q_2^{\{1\}}, \ldots, q_2^{\{\chi\}}\}$. If v_3 reads the incoming channels $\{q_2^{\{1\}}, \ldots, q_2^{\{\chi\}}\}$ in the same order as v_1 writes to the channels $\{q_1^{\{1\}}, \ldots, q_1^{\{\chi\}}\}$, the concatenation of all incoming tokens of v_3 is the same for both the refined process network p_a and the top-level process network p_{org}. Thus, replication preserves the correctness under the described condition.

The correctness of unfolding is shown by considering EPN $e = \langle P, u, l, p_{org} \rangle$. First, we suppose that function l contains all possible channel matches and that all structural expansions defined by function u preserve the input to output relation. In other words, we suppose that function u defines the structural expansion $u(v) = p$ of process v with $p \in P$. Then, process v has the same amount of input and output channels as refinement network p and applying the same input sequence to v and p produces the same output sequence. As unfolding merely replaces processes by their structural description, the refined process network has still the same input to output behavior.

3.4.3 High-Level API

After describing the basic concepts of EPNs and defining the conditions when replication and unfolding preserve the correctness, we are able to extend the DAL programming language API to also support EPNs.

The topology specification of an EPN is composed of the topology of multiple process networks, each specified by the API shown in Listing 2.2. The process element is extended with the attributes *type* and *replicable* as described in Definition 3.1. In addition to the specification of the process networks, the transformation functions u and l have to be defined within the specification of the EPN.

The functionality of the individual processes is described in separate description files written in the high-level programming language illustrated in Listing 2.1. As the structural expansion of a process with a process network does not change the external interface, i.e., the incoming and outgoing channels, the structural expansion is completely transparent towards the functionality of the other processes. However, in case of replication, the actual processes and FIFO channels are not known at specification time. Thus, the source processes of a replicated process have to write to a FIFO channel that is not known when the application is specified and similarly, the sink processes of a replicated process have to read from a FIFO channel that is not known when the application is specified. For this purpose, we propose the API outlined in Listing 3.1 to iterate over all possible FIFO channels. The basic idea is that, per iteration, the source process still writes to and the sink process still reads from one instance of the replicated FIFO channel. We propose that the FIFO channels are addressed by their basename, i.e., the name of the channel before being replicated, and a counter, which is stored in the state of the process. The number of replicas per channel can be obtained from a global variable

List. 3.1: Pseudo code of a process sending data to a replicated FIFO channel (addressed by the output port "out") in order to illustrate the proposed extension of the DAL programming language API to specify EPNs.

```
int FIRE(ProcessData *p) {
    ...
    port_basename = "out";
    port = createport(port_basename, p->fifocounter);
    WRITE(port, buffer, size);
    p->fifocounter = (p->fifocounter + 1) % REPLICATIONS_Q1;
    ...
}
```

that is set automatically during the synthesis step. In Listing 3.1, the synthesizer sets the variable `REPLICATIONS_Q1` to the number of replicas of FIFO channel "q1". In the DAL design flow, the code shown in Listing 3.1 is created by an automated source-to-source code transformation during software synthesis.

Ex. 3.4: *Consider the process network outlined in Fig. 3.3, which has been obtained by replicating process* v_2 *three times. At specification time, process* v_1 *writes to FIFO channel* q_1*, which does not exist anymore in the refined process network. Instead,* v_1 *has three output FIFO channels* $q_1^{\{1\}}$, $q_1^{\{2\}}$*, and* $q_1^{\{3\}}$*. The API outlined in Listing 3.1 hides the transformation details from the system designer who can still use FIFO channel* q_1*.*

In case the functionality of a process is given only by either a behavioral or a structural description, one might obtain the other description by a transformation. For instance, a behavioral description can be obtained from a structural one by implementing the channels as shared variables. Conversely, one might use the techniques described in [KRD00, VNS07] to obtain a structural description out of a behavioral one.

3.5 Application and Mapping Optimization

In this section, we introduce a novel performance analysis model for applications specified as EPNs and introduce a novel optimization heuristic to determine a good abstraction of the EPN application for a given (sub-)architecture. The aim of the heuristic is to identify which application structure and mapping lead to the highest throughput. To this end, we minimize the maximum core utilization for a given invocation interval of the source process. The invocation interval is then adjusted so that the maximum core utilization becomes 100 %. Finally, the new invocation interval is used to calculate the maximum throughput of the EPN application.

3.5.1 Preliminaries and Notation

The target architecture of the considered optimization problem is a sub-architecture, which is formally defined as follows.

Def. 3.3: **(Sub-Architecture)** *A sub-architecture* $\mathcal{A}^s = \langle C, n \rangle$ *consists of a set of cores* C *that are connected by a communication network* n*, e.g., a bus or a NoC. A core* $c \in C$ *is characterized by the cycle time* t_c^0*.*

Furthermore, the binding of processes to processing cores is defined by the assignment function Γ.

Def. 3.4: **(Assignment Function)** *Suppose that KPN* $p_a = \langle V_a^e, Q_a^e \rangle$ *is a valid refinement of the top-level process network* p_{org} *of EPN* e*. Then, the mapping of* p_a *onto sub-architecture* $\mathcal{A}^s = \langle C, n \rangle$ *is defined by the assignment function* $\Gamma(v, c) \in \{0, 1\}$ *that is* 1 *if and only if process* $v \in V_a^e$ *is mapped onto core* $c \in C$*:*

$$\Gamma(v, c) = \begin{cases} 1 & \textit{if } v \textit{ is mapped onto core } c, \\ 0 & \textit{otherwise.} \end{cases} \tag{3.1}$$

In order to guarantee that each task is assigned to exactly one core, the following equation has to be fulfilled for all processes $v \in V_a^e$:

$$\sum_{c \in C} \Gamma(v, c) = 1 \quad \forall v \in V_a^e \tag{3.2}$$

with C the set of processing cores of the considered sub-architecture $\mathcal{A}^s = \langle C, n \rangle$.

3.5.2 Performance Model

Next, we describe a novel performance model that is used in the design space exploration to analyze a candidate network $p_a = \langle V_a^e, Q_a^e \rangle$. The performance model aims to provide a metric for the average utilization of a core so that the maximum average core utilization can be minimized. Table 3.1 summaries the most important performance parameters that are used in the following.

The iterative execution of a behavioral process v is characterized (per invocation of the FIRE procedure) by the number of computation cycles n_v^{co}, the number of read cycles $n_{i,\lambda}^{re}$ per input channel $i \in in$, and the

Tab. 3.1: List of parameters that are used in the performance model.

param.	**description**
$\Gamma(v,c)$	assignment function (1 if and only if process v is mapped onto core c, otherwise 0)
t_c^0	cycle time on core c
n_v^{co}	number of computation cycles of process v
$n_{i,\lambda}^{re}$	number of cycles to read from channel i for communication mode λ
$n_{o,\lambda}^{wr}$	number of cycles to write to channel o for communication mode λ
n_v	total number of cycles per execution of the FIRE procedure of process v
$f_{v,p}^{rel}$	relative execution rate of process v in process network p
f_v^{abs}	absolute execution rate of process v
η_i^{re}	number of readings from channel i per execution of the FIRE procedure
η_o^{wr}	number of writings to channel o per execution of the FIRE procedure
γ_c	context overhead per time instance on core c
T_c^{cont}	context switch time on core c
ν_v^{co}	relative number of computation cycles of process v when it is recursively unfolded
ν_i^{re}	relative number of cycles that process v has to read from channel i when it is recursively unfolded
ν_o^{wr}	relative number of cycles that process v has to write to channel o when it is recursively unfolded

number of write cycles $n_{o,\lambda}^{wr}$ per output channel $o \in out$. The number of read and write cycles depends on the data-volume, thus the average size of the packets and the number of packets that are read or written per execution of the FIRE procedure. The factor λ indicates the dependency of the read and write instructions on the channel's location. For simplicity, we just differ between inter-core and intra-core communication, thus $\lambda \in \{\textit{inter-core}, \textit{intra-core}\}$. However, the concept can be extended to more complex communication topologies, e.g., by differentiating between the number of hops it takes to transmit a packet from source to destination. Therefore, the average number of cycles that process v performs in one invocation of the FIRE procedure is:

$$n_v = n_v^{co} + \sum_{i \in in} n_{i,\lambda}^{re} + \sum_{o \in out} n_{o,\lambda}^{wr}. \tag{3.3}$$

The average utilization of a core also depends on the average execution rates of all processes $v \in V_a^e$. First, we note that the absolute execution rate of process v cannot be specified in advance as it depends on the execution rate of the processes that v has replaced. However, the latter might be known only after the design transformation. Second, we note that a process can occur in multiple process network specifications and that its execution rate might be different for every process network p that v can occur in. Thus, at specification time, we characterize process v by a set of relative execution rates $f_{v,p}^{rel}$ with one execution rate per process network p that v can occur in. In addition, one process $\tilde{v}$ of the top-level process network p_{org} is characterized by an absolute execution rate $f_{\tilde{v}}^{abs}$. This enables us to calculate the absolute execution rates f_v^{abs} of all processes $v \in V_a^e$ after the design transformation step. Later, we use these absolute execution rates to calculate the average utilization of a core. The absolute execution rates can be calculated in a top-down approach following the performed design transformations. The algorithm starts with process $\tilde{v}$ and applies the following rules for any process v:

1. If v belongs to p_{org}, then $f_v^{abs} = f_{v,p}^{rel} \cdot f_{\tilde{v}}^{abs}$.
2. If v belongs to $p \in \{P \backslash p_{org}\}$ and replaces process $\hat{v}$, then $f_v^{abs} = f_{v,p}^{rel} \cdot f_{\hat{v}}^{abs}$.
3. If v is instantiated by replicating $\hat{v}$ χ times, then $f_v^{abs} = \frac{1}{\chi} \cdot f_{\hat{v}}^{abs}$.

If multiple processes share the same resource, the synchronization and scheduling overhead might affect the overall performance. In this work, we differ between event-triggered and time-triggered scheduling policies. Typically, time-triggered scheduling policies cause a constant overhead [HHB+12], while the overhead caused by an event-triggered scheduling policy depends on the workload. Suppose that multiple processes $v = \langle name, type, replicable, in, out \rangle$ share the same processing core. A process can become blocked when reading from an empty input FIFO $i \in in$ or writing to a full output FIFO $o \in out$. A pessimistic assumption for a non-preemptive scheduling policy is that the process is always blocked when reading from and writing to a FIFO channel. Thus, the total average context overhead per time instance on core c is given by:

$$\gamma_c = \left(\sum_{v \in V_a^e} \Gamma(v, c) \cdot f_v^{abs} \cdot \left(\sum_{i \in in} \eta_i^{re} + \sum_{o \in out} \eta_o^{wr} \right) \right) \cdot T_c^{cont}, \tag{3.4}$$

where T_c^{cont} is the context switch time on core c, η_i^{re} is the average number of readings from channel i per execution of the FIRE procedure, and η_o^{wr} is the average number of writings to channel o per execution of the FIRE procedure. Clearly, if only one process is mapped onto core c, there is no context switching overhead and $\gamma_c = 0$.

If a process v is recursively instantiated, the execution time might be reduced with every recursion step. In order to model this reduction in the performance model, a recursive process v is annotated by a relative number of computation cycles ν_v^{co}, a relative number of read cycles ν_i^{re}

per input channel $i \in in$, and a relative number of write cycles ν_o^{wr} per output channel $o \in out$. If process v is recursively unfolded, its number of computation cycles is reduced to $\nu_v^{co} \cdot n_v^{co}$ cycles in every recursion step. Similarly, if the execution time does not change, $\nu_v^{co} = 1$.

Finally, note that the performance analysis model is not restricted to the maximization of the throughput, but can also be applied to optimize other performance metrics such as the energy consumption. In this case, the invocation interval might be fixed and the utilization of the individual cores is used to calculate the average expected energy consumption of the system.

3.5.3 Alternative Process Network Calculation

The goal of the optimization step is to find a process network and a corresponding mapping that maximize the application's throughput on a given sub-architecture. As previously discussed, this is equivalent to minimizing the maximum core utilization. Therefore, the objective function can be stated formally as follows:

$$\min \left\{ \max_{c \in C} \left\{ \gamma_c + \sum_{v \in V_a^e} \Gamma(v, c) \cdot f_a^{v,p_a} \cdot n_v \cdot t_c^0 \right\} \right\}, \tag{3.5}$$

where γ_c is defined as in Eq. (3.4). $\Gamma(v, c)$ has to fulfill the constraint specified by Eq. (3.2) and $p_a = \langle V_a^e, Q_a^e \rangle$ is a valid refinement of the top-level process network of EPN e.

The EPN semantics can be applied to a wide variety of optimization techniques including simulated annealing [KGV83] and Evolutionary Algorithm (EA) as, for instance, used in Chapter 2. In the following, we discuss an alternative optimization approach that uses graph partitioning, which was previously proposed to find good replication degrees [HCK+09], to find a good process network instance and the corresponding mapping. The basic idea of Algorithm 3.3 is to balance the workload between the available processing cores. It first identifies the processing core with the highest utilization and its process v_{work} with the largest amount of work. Then, it virtually adds process v_{work} to all processing cores and selects the processing core with the lowest utilization. It stops if the ratio between highest and lowest utilization is lower than a predefined balance factor that is specified as an input to the algorithm. Selecting a good balance factor might be difficult. However, our experiments have shown that a balance factor of 1.2 generates good results in general.

As extensive unfolding or replication increases communication overheads, Algorithm 3.3 primarily migrates processes to processing cores with low utilization. Process v_{work} is migrated to the processing core with the lowest utilization if the maximum utilization over all processing cores can be reduced. Otherwise, if the maximum utilization cannot be reduced anymore by migrating processes, Algorithm 3.3 unfolds or replicates v_{work}, making the largest indivisible unit of work smaller. In case a process can be both unfolded and replicated, Algorithm 3.3 aims to increase primarily the task and pipeline parallelism by unfolding the process. The new processes are distributed between the processing core of v_{work} and the processing core with the lowest utilization so that both processing cores have a balanced workload. In case that v_{work} can be neither unfolded nor replicated, the algorithm stops or, if there are further processes assigned to the processing core of v_{work}, it migrates these processes to the processing core with the lowest utilization.

The complexity of Algorithm 3.3 depends on the application and the number of processing cores. In fact, the complexity of a single iteration mainly depends on the total number of processing cores and the used sorting technique (Line 3). However, as each loop iteration only changes the utilization of two processing cores, the sorting algorithm can use the result of the previous iteration and just change the position of these two entries. The complexity of an iteration is therefore $O(|C|)$ with $|C|$ the number of processing cores of the considered sub-architecture.

3.6 Application Transformation

In the previous section, we have shown how to calculate the structure of an EPN application so that the throughput of the application is optimized on a given sub-architecture. Therefore, during design space exploration, multiple instances of the same application are synthesized automatically and each instance is optimized for a different number of available cores. The runtime manager uses these information to select the best suited instance for the available computing resources. In case that the computing resources available to a certain application change at runtime, e.g., as other applications are started or stopped, the runtime manager must transform the application into another implementation without discarding the program state of the application.

In this section, we describe a transformation technique to do so. First, we illustrate the considered problem and the approach to solve it. Then, we discuss the characteristics that must hold for applications that can be transformed into an alternative instance. Finally, we detail the proposed transformation technique.

Algorithm 3.3 Calculating a process network and its corresponding mapping so that the throughput of the EPN application $e = \langle P, u, l, p_{org} \rangle$ is maximized on sub-architecture $\mathcal{A}^s = \langle C, n \rangle$.

```
Input: EPN $e = \langle P, u, l, p_{org} \rangle$, processing cores $C$, balanceFactor
Output: process network and mapping
addAllProcessesToASingleCore($p_{org}$)
while True do
    sortCoresByUtil($C$)                ▷find core with maximum utilization
    maxCore ← CoreWithMaxUtil($C$)
                                        ▷find process with largest computing demand
    process ← largestProcess(maxCore)
                                        ▷find core with lowest utilization after adding process
    minCore ← CoreWithMinUtilAfterAdding($c$, process)

                                        ▷check the overall balance of the system
    if util(maxCore) < util(minCore) $*$ balanceFactor then
        finish()
    end if

                                        ▷migrate process to minCore or refine the process
    if utilAfterAdding(minCore, process) < util(maxCore) then
        addTo(process, minCore)         ▷migrate process
        removeFrom(process, maxCore)
    else if process can be refined then      ▷refine process and . . .
                                        ▷. . . distribute processes uniformly to maxCore, minCore
        removeFrom(process, maxCore)
        subProcesses ← refine(process)  ▷unfold or replicate process
        distributeAndAdd(subProcesses, maxCore, minCore)
    else if process is the only process assigned to maxCore then
        finish()                        ▷no more refinements are possible
    else                                ▷migrate remaining processes of maxCore to minCore
        remaining ← removeAllExceptOneFrom(process, maxCore)
        addTo(remaining, minCore)
    end if
end while
```

3.6.1 Problem Description

Transforming a stateful application from one process network into another process network requires the specification of how the application's state is transferred from one implementation into another one. This is usually not trivial, as the following example shows.

Ex. 3.5: *Assume that an application has two implementations with different degrees of parallelism. Implementation 1 consists of process* v_1 *and implementation 2 consists of processes* v_2 *and* v_3 *that are connected by a FIFO channel q. Suppose now that the application is transformed from implementation 1 to implementation 2. One possible way to do so is as follows:* v_1 *is stopped immediately and the new processes and channels are installed. To maintain the program state, a transformation procedure would be used that takes as input the program counter and all variables of* v_1 *and generates the program counters and the variables of* v_2 *and* v_3*, as well as the content of channel q. Once variables and program counters have been assigned,* v_2 *and* v_3 *are started. Programming such a procedure is typically complicated and it is even more laborious to derive a procedure that performs the opposite operation, i.e., generates the variables and the program counter of* v_1*. However, such a procedure is needed to transform the application back to implementation 1, e.g., if the system enters again the previous execution scenario.*

In the following, we only consider the case where a process network has been obtained by using the refinement strategy *unfolding*. In other words, the top-level process network of an EPN can be refined only by replacing processes by their refinement network. However, in the case we are now considering, replication can be modeled by replacing the process by a refinement network that consists of a fork process, multiple replicas of the process, and a join process.

To tackle the previously describe challenge, we propose the following solution to transform an application from a process network into an alternative process network:

- We transform the application stepwise into the alternative process network. In each step, either we refine a process, i.e., we replace it by its refinement network, or we replace the processes and channels of a refinement network by their origin process. We call the first operation *expansion* and the second one *contraction*.
- We restrict the points in time for expansion and contraction: A process / refinement network must reach a normal state in order to be expanded / contracted.
- We describe a scheduling strategy that brings a process or refinement network to a normal state.
- We extend the API proposed in Section 3.4 by two procedures that transform the state of a process into the state of its refinement network, and vice versa. Due to these procedures, a stateful process can be replaced transparently by its refinement network.

3.6.2 Execution Model

The goal is to come up with an execution model that requires the system designer to specify only two additional transformation functions, namely one for the expansion and one for the contraction. However, the system designer should not have to deal with program counters and implicit state that is on the stack when designing the transformation functions. Therefore, we restrict the points in time for the expansion and the contraction: A process can only be expanded if it has finished its FIRE procedure and a process network can only be contracted if all of its processes have finished their FIRE procedure and if its internal channels contain a statically specified number of tokens.

In the following, we discuss the considered execution model individually for channels, processes, and refinement networks. Based on that, we define the characteristics that must hold for processes that can be refined by a refinement network and for process networks that act as refinement networks so that the above stated goal can be achieved. This forms the basis for a novel transformation technique that is proposed in Section 3.6.3 and transforms a process network transparently into an alternative process network.

Channel

A channel q of network p contains valued tokens that are read and written in FIFO order. The number and values of the tokens determine the state $\pi_q \in \Pi_q$ of channel q whereby Π_q is the set of channel states of q that are admissible in any correct execution trace of p.

Process

During the execution of its FIRE procedure, process v reads tokens from its input channels and writes tokens to its output channels, thereby modifying its state π_v taken from the set of admissible states Π_v of v. As previously motivated, v can only be expanded if its FIRE procedure has reached its end and is not yet re-started. We call any admissible state of v where v has finished its FIRE procedure a *normal state* of v.

Def. 3.5: *Assume that process v is a node of a process network p_a and v either is part of a refinement network or can be replaced by a refinement network. Then, the following characteristics must hold for v and p_a:*

1. *(BOUNDEDNESS) Assume that the input channels of v contain a finite number of tokens. Then, the execution of v is finite, i.e., after a bounded number of executions, its FIRE procedure is blocked on reading from an empty channel.*

2. (TERMINATION) *There exists a constant L such that for any admissible state of v and for any admissible states of the input channels of v with L tokens each, the* FIRE *procedure terminates, i.e., it reaches its end.*

3. (DEADLOCK FREE) *v can infinitely often execute its* FIRE *procedure, i.e., network p_a does not contain a deadlock concerning v.*

4. (NO DEAD INPUT CHANNEL) *During each correct execution trace of network p_a of infinite length, an unbounded number of tokens is written into each input channel of v.*

Note that the first two conditions guarantee that the FIRE procedure never runs into an infinite execution, i.e., the FIRE procedure is not allowed to enter an infinite loop.

Refinement Network

A refinement network p has a state π_p, which consists of the states of all its processes and internal channels. A state $\pi_p \in \Pi_p$ is admissible if it appears with a legal input sequence of p, whereby Π_p is the set of admissible states of p.

Refinement network p can only be contracted by its origin process if all of its processes have finished their FIRE procedure and each of its internal channels contains a statically specified number of tokens. We call these numbers the *normal token distribution* of p and any state of p that fulfills the above stated conditions a *normal state* of p.

In order to replace a process v transparently by its refinement network $p = u(v)$ (and vice versa), the system designer has to define how the state is passed from v to p and from p to v. She does that by specifying the two transformation functions EXPAND and CONTRACT. The EXPAND function E_v maps any normal state π_v of v to a normal state $\pi_p = E_v(\pi_v)$ of p. The CONTRACT function C_p maps any normal state π_p of p to a normal state $\pi_v = C_v(\pi_p)$ of v. Using the above notation, we can define the required characteristics of a refinement network.

Def. 3.6: *The following characteristics must hold for a refinement network $p = u(v)$ of process v:*

1. (BOUNDEDNESS) *Assume that the input channels of p contain a finite number of tokens. Then, the execution of p is finite, i.e., after executing the* FIRE *procedures of its processes a bounded number of times, each of its* FIRE *procedures is blocked on reading from an empty channel.*

2. (SYNTACTICAL EQUIVALENCE) *For each input and output channel of v, there is a corresponding input and output channel of p.*

3. (FUNCTIONAL EQUIVALENCE) *Assume that the input channels of v and p have the same channel state, i.e., the same finite number of tokens with the same values.*
    - *If v is initially in a normal state π_v (i.e., the* FIRE *procedure reached its end, but is not yet re-started), if the state of p initially satisfies $\pi_p = E_v(\pi_v)$, and if v and all processes of p then execute their* FIRE *procedure iteratively until being blocked on reading from an empty channel, the sequences of tokens written by v and p to the corresponding output channels are the same.*
    - *If p is initially in a normal state π_p (i.e., the* FIRE *procedures of all its processes are finished and each of its internal channels contains a statically specified number of tokens), if the state of v initially satisfies $\pi_v = C_v(\pi_p)$, and if v and all processes of p then execute their* FIRE *procedure iteratively until being blocked on reading from an empty channel, the sequences of tokens written by v and p to the corresponding output channels are the same.*
4. (REACHABILITY) *There exists a constant K such that for any normal state of p and for any admissible state of the input channels of p with K tokens each, there exists an ordering of complete executions of the* FIRE *procedures*[1] *of the processes of p such that p has again a normal token distribution. Furthermore, the* FIRE *procedure of every process in p is executed at least once in such an ordering.*

These conditions guarantee that both the expansion of process v by refinement network $p = u(v)$ and the contraction of p by v do not change the functionality of the whole process network. Furthermore, the reachability condition states that there is at least one schedule that brings network p from a normal state to another normal state.

In many situations, the above stated conditions do not impose severe restrictions. Consider, for example, a process of a video processing application. Its functionality can often be split into sub-steps or rewritten such that it first splits large data blocks into smaller data blocks and then operates on these tokens. Both described refinements do not modify the functionality, the execution is still bounded, and, if enough tokens are available in their input channels, the refinement networks can be scheduled so that they enter a normal state. Finally, note that the discussed model of computation is still more general than the SDF [LM87] model of computation that has been applied in previous works to adapt the application structure.

---

[1] The execution of a single FIRE procedure does not have to be atomic; it can be interrupted by another FIRE procedure at any time.

Extension of the High-Level API for EPNs

Based on the above discussed considerations, we propose the following extension of the high-level API described in Section 3.4 to specify the additional transformation functions. For each process that can be refined, the system designer has to specify the procedures EXPAND and CONTRACT. Listing 3.2 illustrates the proposed extension. On the one hand, procedure EXPAND implements function E_v and generates the state of the refinement network. On the other hand, procedure CONTRACT implements function C_v. It reads the remaining tokens from all internal channels of the refinement network (the number of tokens is statically specified by the normal token distribution). Then, it generates the process state. It is the system designer's responsibility to ensure that the characteristics stated in Definitions 3.5 and 3.6 hold for all processes and refinement networks.

List. 3.2: Example of an implementation of the procedures EXPAND and CONTRACT for process v_1, which has the refinement network $p = \langle V = \{v_2, v_3\}, Q = \{q_1\}\rangle$.

```
// generate process state of v2 and v3, and write initial
   tokens
void expand(StateV1 *V1, StateV2 *V2, StateV3 *V3,
      Channel *Q1) {
  V2 = generateStateOfV2(V1);
  V3 = generateStateOfV3(V1);
  writeInitialTokens(Q1, V1);
}

// generate process state of v1
void contract(StateV1 *V1, StateV2 *V2, StateV3 *V3,
      Channel *Q1) {
  channelState = readChannelState(Q1);
  V1 = generateStateOfV1(V2, V3, channelState);
}
```

3.6.3 Transformation Technique

So far, we have defined the characteristics that must hold for processes that can be refined and for process networks that can act as refinement network. Next, we detail the actual transformation technique.

The basic idea of the transformation technique is to transform an application stepwise into an alternative process network. In each step, it either expands or contracts one process. In the following, we will discuss first the case of expanding a process and then the case of contracting a process.

Expanding a Process

Assume that process v is a node of network p_a. It follows from Definition 3.6 that the expansion of v by the refinement network $p = u(v)$ does not change the functionality of p_a if v is expanded after finishing its FIRE procedure and if the state of p is initially $\pi_p = E_v(\pi_v)$, whereby π_v is the state of v after finishing the FIRE procedure. Therefore, a prerequisite for the expansion is that v finishes its FIRE procedure. However, it follows from the characteristics stated in Definition 3.5 that v will do that if p_a is executed long enough. Algorithm 3.4 summarizes the steps to expand a process.

Algorithm 3.4 (EXPANSION) Replace process v by its refinement network $p = u(v)$.

```
install all processes and channels of $p$
connect the channels of $p$ to the corresponding processes of $p$
stop process $v$ at the end of its FIRE procedure
use the EXPAND procedure of process $v$ to generate the process states
    of all processes of $p$ and the initial tokens of all channels of $p$
connect incoming and outgoing channels of $v$ to the corresponding
    processes of $p$
start all processes of $p$ and remove process $v$
```

Contracting a Process

Assume again that process v is a node of process network p_a and that v has the refinement network $p = u(v)$. It follows from Definition 3.6 that contracting the refinement network $p = u(v)$ by v does not change the functionality of p_a if p is stopped in a normal state π_p and if the state of v is initially $\pi_v = C_v(\pi_p)$. p is in a normal state if all of its processes have finished their FIRE procedure and if each internal channel contains a statically specified number of tokens, also known as the normal token distribution.

Therefore, a prerequisite for the contraction is that the refinement network p is in a normal state. However, if the FIRE procedures of its processes are executed iteratively in a greedy manner, p might never enter such a state. Thus, in the following, we will first describe a scheduling strategy that executes the processes of network p_a such that the refinement network p enters a normal state. The basic idea of the scheduling strategy shown in Algorithm 3.5 is that each process of the refinement network observes the number of tokens in its input and output channels and only starts its FIRE procedure if certain conditions are fulfilled. In fact, if a

Algorithm 3.5 Scheduling strategy to bring a refinement network p being part of process network p_a to a normal state.

```
01 execute all processes of $p_a$ except those of $p$ in a greedy manner,
   i.e., execute their FIRE procedures iteratively
02 execute all processes of $p$ in a greedy manner. However, do only
   restart the FIRE procedure of a process $v$ in $p$ if at least one of the
   following conditions holds:
03 • $v$ has an internal input channel that contains more tokens than its
     normal number of tokens,
04 • $v$ has an internal output channel that contains less tokens than its
     normal number of tokens, or,
05 • the FIRE procedure of another process in $p$ is directly or indirectly
     blocked on an output channel $q$ of $v$. A process is directly blocked
     on $q$ if the process is the reader process of $q$. A process is indirectly
     blocked on $q$ if the process is blocked on an input channel of $p$
     whose writer process itself is blocked by some other process and if
     this chain finally ends in $q$
06 stop if all processes of $p$ have finished their FIRE procedures and no
   FIRE procedure can be restarted
```

channel has more tokens than its normal number of tokens (defined by the normal token distribution), the reader process continues its execution. The writer process continues its execution if a channel has less tokens than its normal number of tokens.

Note that the strategy of Algorithm 3.5 must only be used to schedule the network when a particular refinement network is supposed to be contracted. Example 3.6 emphasizes the role of Line 5, which is used to resolve deadlocks that are imposed by the scheduling strategy (the processes of the refinement network might be blocked artificially, i.e., they cannot necessarily restart their FIRE procedure).

Ex. 3.6: *Assume that process $\tilde{v}$ of refinement network p must restart its* FIRE *procedure as either the rule on Line 3 or the one on Line 4 holds for one of its input or output channels. Then, $\tilde{v}$ may block on reading from another input channel that does not contain enough tokens and whose writer process is also blocked. If the writer process is also in p, the* FIRE *procedure of $\tilde{v}$ is directly blocked on an output channel of some process v in p. On the other hand, if the writer process is not in p, it must again be blocked by some other process as only processes in p can be blocked artificially. In fact, the thereby created chain must end at some process v in p so that $\tilde{v}$ is indirectly blocked on an output channel of v and the block can only be resolved if v restarts its* FIRE *procedure.*

Thm. 3.1: *Assume that network p is a refinement of process v, part of process network p_a, and the characteristics stated in Definitions 3.5 and 3.6 hold for v, p, and p_a. After executing the* FIRE *procedures of its processes for a finite number of times, network p_a including p is scheduled according to the rules stated in Algorithm 3.5. Then, p will eventually enter a normal state.*

Proof. We know that we originally replaced in network p_a process v by a correct refinement network $p = u(v)$. Due to Definitions 3.5 and 3.6, replacing v by p did not change the functionality of p_a so that no deadlock can occur in p if the FIRE procedures of all processes of p_a are executed iteratively. Given this property, the basic idea of the proof is to observe the sequences of tokens in the input channels of v in the origin network p_a with v instead of p. Then, we use these sequences to determine a schedule for the processes of p. Unless this schedule leads to a deadlock, p can be executed according to this schedule even if p is embedded into p_a.

Let us first consider the point in time when we started to schedule p_a by Algorithm 3.5. Assume that the highest number of executions of a FIRE procedure in any process of p was f, i.e., no process in p executed the FIRE procedure more than f times. As all processes of p_a are deterministic, the sequences of tokens in the input channels of p are independent of the execution order of the processes. Due to the functional equivalence of the expansion, the sequences are the same as that of the origin network p_a with v instead of p. Let us observe the input channels of v and the corresponding sequences that would have occurred if we had not done the expansion, starting from the instance of the expansion. As v has no dead input channels (see Definition 3.5), we stop the observation if each sequence contains at least $f \cdot K$ tokens whereby K is defined as in the reachability condition of Definition 3.6.

Let us go back to the refined network. We know from the reachability condition in Definition 3.6 that starting from any normal state of p, there exists an ordering of complete executions of the FIRE procedures of the processes of p such that p is again in a normal state. In such a sequence, each process in p executes its FIRE procedure at least once. Now assume that we go from one such state to the next one f times. Clearly, there would be enough tokens in the input channels to allow for this schedule and the FIRE procedure of each process in p must be executed at least f times. As defined above, no process in p executed its FIRE procedure more than f times when starting to schedule p_a by Algorithm 3.5. Therefore, in order to reach the final normal state from the current state, there are executions of the FIRE procedures still left for some processes, but no process executed its FIRE procedure more often than necessary in order to reach the final normal state (after f iterations).

As all processes of p_a are deterministic, the ordering of executing the FIRE procedures does not matter. In other words, if we start from any state and execute the FIRE procedures by a certain scheduling method a given number of times, then we can reach the same state by any other scheduling method provided that we do not execute the FIRE procedure more often than this number of times. However, no online scheduling strategy knows the number of times that the FIRE procedures should be executed so that network p reaches the final normal state. Therefore, in order to prove that network p enters a normal state if p_a is scheduled according to the rules of Algorithm 3.5, we have to show that the scheduling strategy *a*) does not execute the FIRE procedure of a process in p more often than the number of times that is necessary to reach the final normal state and *b*) does not lead to deadlocks, whereby network p initially started in a normal state and a greedy scheduler with an upper bound on the number of FIRE executions for each process in network p would enter the final normal state.

First, we show that Algorithm 3.5 only starts a FIRE procedure if the greedy scheduler would also do so. Assume that an internal channel contains a smaller number of tokens than its normal number, then the writing process needs to execute its FIRE procedure at least once. Assume that an internal channel contains a larger number of tokens than its normal number, then the reading process needs to execute its FIRE procedure at least once. If the FIRE procedure in one of the above mentioned cases is blocked due to an internal input channel, the process writing to this channel must also execute its FIRE procedure. Clearly, Algorithm 3.5 covers these cases.

Now, let us show that the scheduling strategy of Algorithm 3.5 does not lead to a deadlock. Assume towards a contraction that there is a deadlock in p, i.e., the number of tokens in the internal channels does not yet correspond to the normal token distribution, but no process can proceed anymore. Consequently, at least one process in p is blocked on reading from an internal or input channel of p and the remaining processes of p have completed their FIRE procedure, but are not eligible to restart it. However, the case that a process is blocked on reading from an internal channel is resolved as the process that causes the blocking starts its FIRE procedure. In case that a process is blocked on an input channel of p, the execution will only block forever if the writing process (which is not part of p) is blocked itself and a process of p connected to an output channel of p must execute its FIRE procedure at least once more to resolve this blocking. However, this case is resolved as a process that causes an indirect blocking on an output channel restarts its FIRE procedure. □

The steps to contract a refinement network are summarized in Algorithm 3.6.

Algorithm 3.6 (CONTRACTION) Replace the processes and channels of refinement network $p = u(v)$ by process v.

```
install process $v$
use the strategy of Algorithm 3.5 to stop the refinement network $p$ in a normal state
use the COARSEN procedure of process $v$ to generate the process state of $v$
connect the input and output channels of $p$ to process $v$
remove all processes and channels of $p$, and start process $v$
```

Process Migration

Clearly, transforming an application into an alternative process network might also involve the migration of some processes to different processing cores. In the following, we summarize a technique to do so. It migrates a process adhering to the API described in Section 3.4 from one processing core to another one.

Algorithm 3.7 shows the pseudo-code to migrate an individual process from one processing core to another one on a platform with distributed memory. The basic idea of the algorithm is to have a hand-shaking protocol that informs the process to be migrated, v_{mig}, that no more incoming tokens are generated. Afterwards, all in-flight tokens (tokens written by a parent process, but not yet received by the child process) are collected so that process v_{mig}, as well as all incoming and outgoing channels of v_{mig}, can be migrated.

3.7 Evaluation

In this section, we present case studies demonstrating the effectiveness of the EPN semantics targeting two many-core platforms. In the first two case studies, we discuss the effect of replication and unfolding on the throughput of an application. Then, we measure the performance of the proposed application and mapping optimization algorithm. Finally, we evaluate the costs of transforming an application into an alternative process network and investigate how its performance is affected during the transformation.

3.7.1 Experimental Setup

In the following, we describe the hardware setup and the considered benchmark applications.

Algorithm 3.7 Pseudo-code to migrate process v_{mig} of process network p from processing core c_{src} to processing core c_{dst}.

```
                                ▷stop $v_{mig}$, pause parent and child processes of $v_{mig}$
stop process $v_{mig}$ before it starts a new firing
for all $q = \langle v_{src}, v_{dst} \rangle$ s.t. $v_{src}$ <> $v_{mig}$ and $v_{dst}$ == $v_{mig}$ do
    pause process $v_{src}$
    wait until all in-flight tokens of $q$ arrived at destination
end for
for all $q = \langle v_{src}, v_{dst} \rangle$ s.t. $v_{src}$ == $v_{mig}$ and $v_{dst}$ <> $v_{mig}$ do
    pause process $v_{dst}$
    wait until all in-flight tokens of $q$ arrived at destination
end for

install process $v_{mig}$ on $c_{dst}$                          ▷move process to new core
move process state $\pi_{v_{mig}}$ from $c_{src}$ to $c_{dst}$
remove process $v_{mig}$ on $c_{src}$

                                ▷re-instantiate incoming and outgoing channels
for all $q = \langle v_{src}, v_{dst} \rangle$ s.t. $v_{src}$ <> $v_{mig}$ and $v_{dst}$ == $v_{mig}$ do
    install channel $q$ between $c_{v_{src}}$ and $c_{dst}$
                                ▷with $c_{v_{src}}$ being the core of $v_{src}$
    transfer tokens from old to new instance of $q$
    remove old instance of $q$
    resume process $v_{src}$
end for

for all $q = \langle v_{src}, v_{dst} \rangle$ s.t. $v_{src}$ == $v_{mig}$ and $v_{dst}$ <> $v_{mig}$ do
    install channel $q$ between $c_{dst}$ and $c_{v_{dst}}$
                                ▷with $c_{v_{dst}}$ being the core of $v_{dst}$
    transfer tokens from old to new instance of $q$
    remove old instance of $q$
    resume process $v_{dst}$
end for
start process $v_{mig}$ on $c_{dst}$                    ▷re-start the process on target core
```

Hardware Setup

In order to test the effectiveness of the EPN semantics, we extend the prototype implementation of DAL, which is described in Section 2.6, to also support applications specified according to the EPN model of computation. The runtime manager described in Section 2.5.2 has been extended with the ability to transform an application from one process network into an alternative process network according to the technique discussed in Section 3.6.

Two many-core architectures are used as target platforms, namely an Intel Xeon Phi processor and an Intel SCC [HDH+10] processor. Though not designed for embedded applications, for the purpose of the experimental evaluation, they are representatives of future many-core SoC architectures.

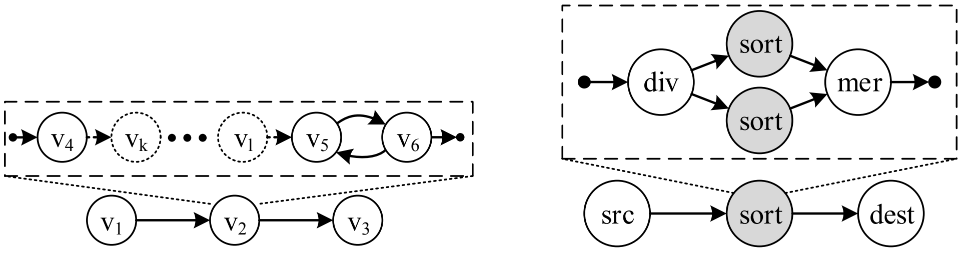
- The Xeon Phi has 60 physical cores (the hyper-threading capability is not used), which are clocked at 1053 MHz, and hosts a Linux operating system with kernel 2.6.38.8. In all experiments, the compiler is ICC 14.0.1 with optimization level O2.
- The Intel SCC processor is configured to run the cores at 533 MHz and both the routers and the DDR3 RAM at 800 MHz. A Linux image with kernel 2.6.32 has been loaded onto each core and the RCKMPI library [URK11] has been configured to use the default channel implementation, i.e., the SCCMPB channel. In all experiments, the compiler is ICC 8.1 with optimization level O2.

In order to identify the application structure and mapping that maximize together the throughput of an application on a given sub-architecture, the following approach is conducted. First, the performance parameters listed in Table 3.1 are obtained by running benchmark configurations on the target platforms. Afterwards, the application structure and the mapping have been optimized using Algorithm 3.3. In fact, the balance factor of Algorithm 3.3 is set to 1.2.

Benchmark Applications

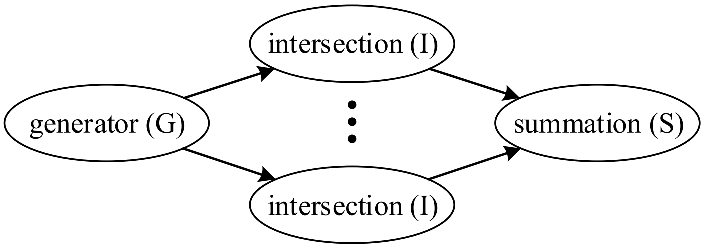
In this section, we analyze the performance of four benchmarks shown in Fig. 3.5.

- **Synthetic.** The synthetic application has a top-level process network with three processes. Process 'v_2' has a refinement network with a variable number of processes.

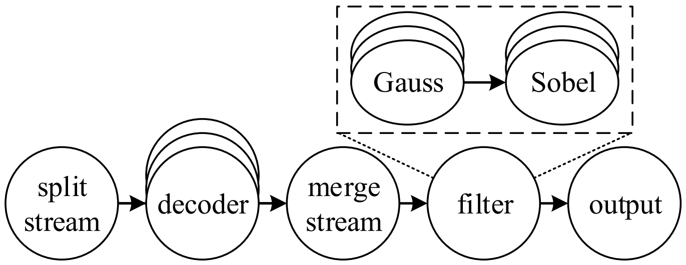


(a) Synthetic application.

(b) Sorting application.



(c) Ray-tracing application.



(d) Video-processing application.

Fig. 3.5: Benchmark applications to evaluate the capabilities of the EPN semantics.

- **Sorting.** The application uses quicksort to sort arrays of variable length. Process 'src' generates the input array, process 'sort' sorts the elements, and process 'dest' displays the output array. As the quicksort algorithm recursively sorts the array, process 'sort' can be replaced by a structural description. 'div' partitions the array into two groups: the first group contains the elements that are smaller than the median value and the second group contains the remaining elements. The divided arrays are sent to a different instance of the 'sort' process. Each 'sort' process sorts the received sub-array. Finally, the sorted sub-arrays are merged into a single array.

- **Ray-Tracing.** The ray-tracing algorithm can have multiple 'intersection' processes to analyze multiple rays concurrently. In addition, the 'generator' process generates the rays and the 'summation' process merges the rays into a single image.

- **Video-processing.** The video-processing application first decodes a MJPEG video stream and then applies several filters to the decoded frames. The 'decoder' process can be replicated, but it cannot be unfolded. The 'filter' process is composed of a Gaussian blur and a gradient magnitude calculation using Sobel filters. If unfolded, both processes can be replicated individually. A gray scale video of 320 × 240 pixels is decoded and analyzed in all evaluations.

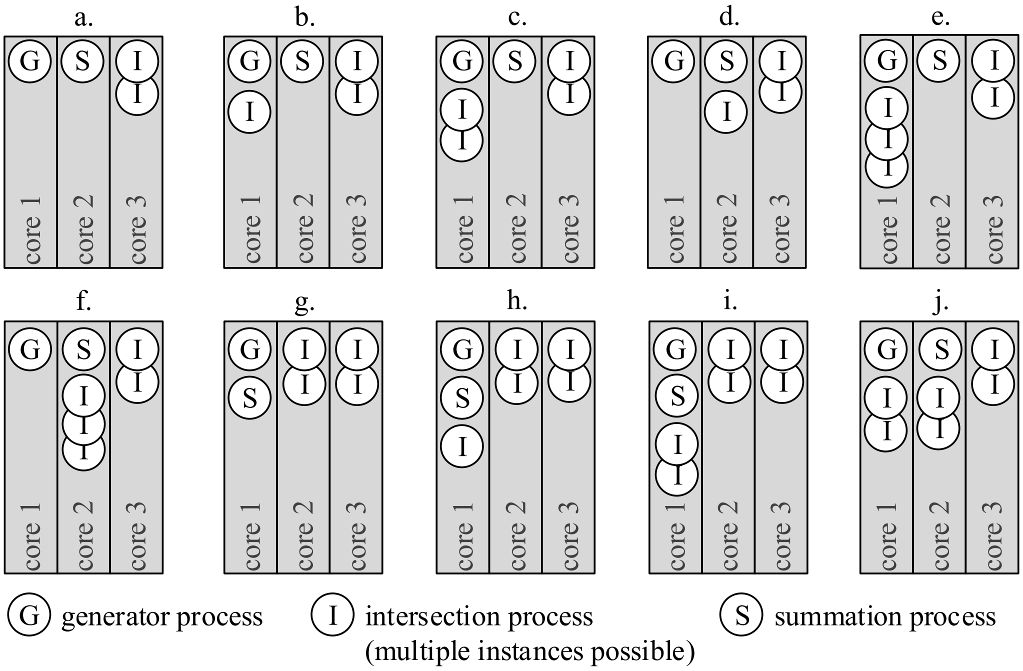
3.7.2 The Optimal Degree of Parallelism

In the first case study, we discuss the effect of replication on the execution time of an application. To this end, we consider the ray-tracing algorithm and compare the execution time of different replication degrees and mappings both when the execution time is estimated with the performance model introduced in Section 3.5 and when the time is measured on Intel's SCC processor.

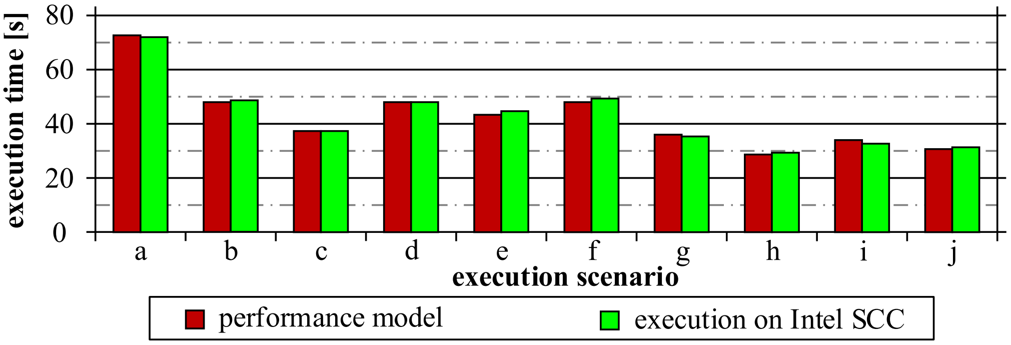
The ray-tracing algorithm is running on three cores of the Intel SCC processor. Figure 3.6a outlines 10 different mapping and replication scenarios of the application. Bubbles indicate the mapping of processes onto processing cores. For example, a bubble with a "G" on core 1 indicates that the 'generator' process is mapped onto core 1. When measuring the execution time on Intel's SCC processor, each core has been selected from a different tile to reduce timing variations due to the network.

To study how replication affects the execution time, the time to generate an image of 100 × 100 pixels, with 10 samples per pixel, has been measured for the scenarios outlined in Fig. 3.6a. Figure 3.6b reports the time that was measured when the scenarios have been executed on the SCC and the time predicted by the performance model described in Section 3.5. Out of all 10 scenarios, the execution time is reduced the most in scenario h, which balances the execution time best between the cores. Thus, in this example, the optimal degree of parallelism is achieved with five 'intersection' processes. The longest execution time is observed in scenario a, which has only two instances of the 'intersection' process. The execution time of the ray-tracing algorithm in scenario h is 29.5 s on the real platform, which corresponds to a speed-up of 2.43 compared to scenario a, which has an execution time of 72.0 s on the real platform.

The average absolute difference between the prediction based on the performance model and the measurement on the real platform is 0.6 s. This indicates that the performance model is able to predict the average execution time accurately if replication is used to improve the parallelism of an application.



(a) Mapping and replication scenarios. “G” refers to the ‘generator’ process, “S” to the ‘summation’ process, and “I” to an instance of the ‘intersection’ process.



(b) Execution time calculated with the performance analysis model described in Section 3.5 vs. the measured execution time on Intel’s SCC processor.

Fig. 3.6: Execution time of the ray-tracing algorithm outlined in Fig. 3.5c for different mapping and replication scenarios. The algorithm is running on three cores of the Intel SCC processor.

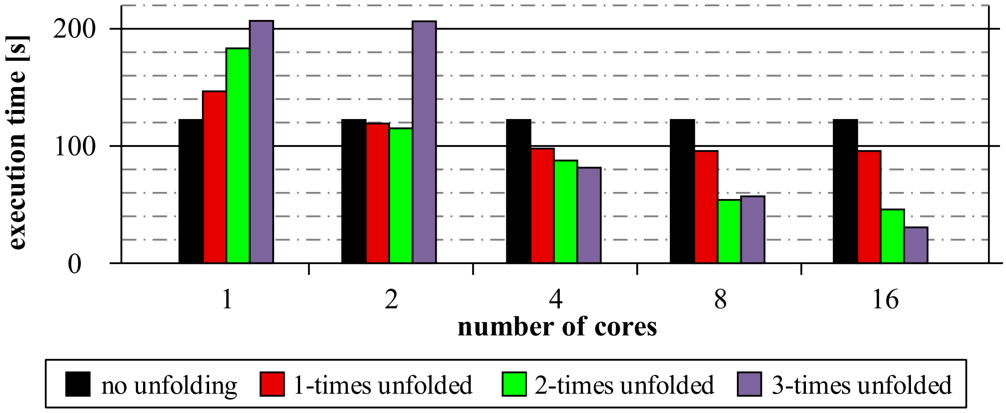


Fig. 3.7: Execution time of the quicksort algorithm for a varying number of available cores and different recursion depths. *No unfolding* refers to the basic quicksort algorithm, i.e., to the top-level process network. *x-times unfolded* refers to an implementation where the 'sort' process is unfolded x times.

3.7.3 Speed-up due to Recursion

To study the effect of recursive unfolding on the execution time of an application, we evaluate the performance of the sorting application on Intel's SCC processor. By unfolding the 'sort' process recursively, the original topology of the sorting application can be refined to have more tasks. As the length of the array that each 'sort' process has to sort is halved in each recursion step, the execution time is reduced with each recursion step, as well.

In Fig. 3.7, the execution time to sort 5′000 random arrays, each with 5′000 elements, is compared for a varying number of available cores and different recursion depths. *No unfolding* refers to the basic quicksort algorithm, i.e., to the top-level process network. *x-times unfolded* refers to an implementation where the 'sort' process is unfolded x times. The evaluation shows that the performance of the sorting algorithm highly depends on the selected recursion depth. On the one hand, if the number of cores is small, a low recursion depth achieves the best performance as the switching and communication overhead is low. On the other hand, if the number of cores is large, a high recursion depth accomplishes a lower execution time as the array is sorted in parallel.

It is worth to note that the speed-up is much lower than the optimal speed-up. The maximum achievable speed-up is 2.3 when using eight cores instead of one core and 4.1 when using sixteen cores instead of one core. This might be because additional time is spent to first split the array into two groups and then to transmit the intermediate results between the different cores.

3.7.4 Application and Mapping Optimization

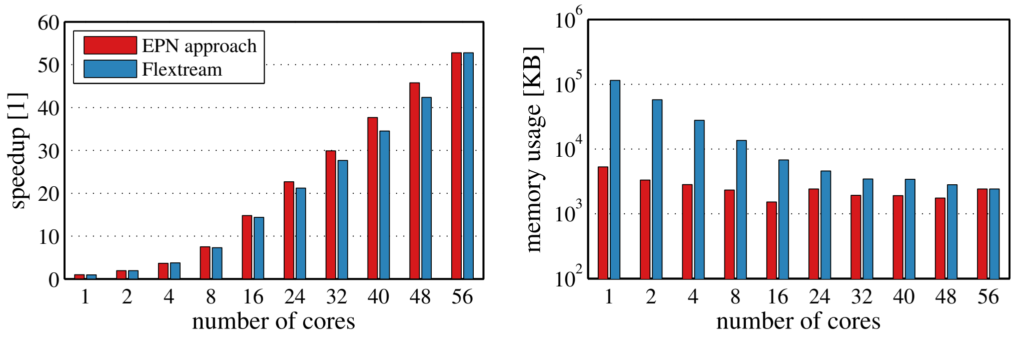
Next, we investigate the question whether a runtime-system that uses the EPN model of computation outperforms a runtime-system that uses a static model of computation in terms of throughput and memory usage.

To this end, we compare the performance of the proposed EPN approach with that of Flextream [HCK+09]. Flextream refines the process network at compile time by using the largest possible resource allocation as target platform. At runtime, it uses this granularity independent of the available resources. In fact, it just assigns the processes that have been originally assigned to cores that are not available, to the remaining cores. For comparability, we extended Flextream's optimization algorithm to also support applications specified according to the EPNs model of comptuation. On the other hand, the EPN approach calculates a different process network and mapping for all possible number of available processing cores.

Figure 3.8 plots the performance of the video-processing application when executing it on the Xeon Phi processor with the number of available cores being varied from one to 56. The speed-up versus the throughput of the top-level process network executed on one core is shown in Fig. 3.8a. The speed-up achieved with the EPN approach is up to 10 % higher than that achieved with Flextream. The EPN approach achieves speed-ups close to the theoretical maximum for any investigated number of available cores. The speed-ups achieved with Flextream can be lower than that of the EPN approach if the processes cannot be evenly distributed among the available cores. This effect is particularly evident for 48 cores. The EPN approach does not suffer from this effect, as it adapts the application's degree of parallelism.

As the EPN approach adapts the number of processes to the number of available cores, the average memory usage per core is almost constant with the EPN approach, see Fig. 3.8b. It is up to 22.5 x less than with Flextream, which uses the same process network for all possible resource allocations.

We conclude that the EPN approach outperforms a static model of computation in terms of throughput and memory usage when the available computing resources are not known at design time or may change at runtime. Compared with Flextream, the EPN approach achieves up to 10 % higher throughput and has up to 22.5 x less memory usage. The EPN approach mainly benefits from the better suited application parallelism that reduces inter-process communication and scheduling overheads.



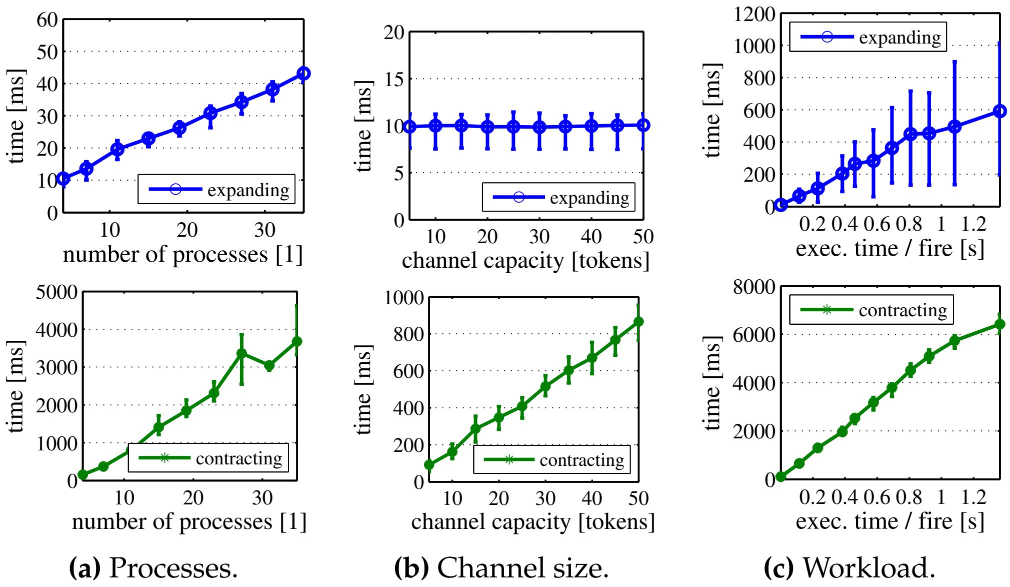
(a) Speedup. **(b)** Memory usage per core.

Fig. 3.8: Performance of the video processing application for different resource allocations. The EPN approach calculates a different process network and mapping for all possible number of available processing cores using Algorithm 3.3. Flextream optimizes the process network for the largest possible resource allocation. If less processing cores are available, it assigns the processes that have been originally assigned to cores that are not available, to the remaining cores.

3.7.5 Transformation Costs

To evaluate the costs of transforming an application into an alternative process network, we measure the time to expand and coarsen process v_2 of the synthetic application that is shown in Fig. 3.5a. When measuring the transformation time, we vary either the number of processes of the refinement network, the capacity of the channels, or the workload of processes v_2 and v_5. If not varied, the refinement network has three processes, the channels have a capacity of five tokens, and a process just reads one token from all of its input channels and writes one token to all of its output channels. For brevity, we only report the results for the Intel Xeon Phi processor; however, the results for the Intel SCC processor exhibit similar trends.

Figure 3.9 shows the time to expand and coarsen process v_2. The reported numbers are the mean of 70 runs and the bottom and top of the error bars are the 16-th and 84-th percentile. The time to expand process v_2 increases linearly with the number of processes of the refinement network and highly depends on the work performed per invocation of the FIRE procedure. However, the time is independent of the channel capacity as there is always only one process involved in the expansion. The time to coarsen process v_2 increases linearly with all investigated parameters. In fact, coarsening takes up to ten times longer than expanding, mainly as the procedure of bringing a refinement network to an admissible state requires more steps than the procedure of bringing a single process to an admissible state.



(a) Processes. **(b)** Channel size. **(c)** Workload.

Fig. 3.9: Time to expand and coarsen the process network of the synthetic application that is shown in Fig. 3.5a.

Next, we measure the time to expand, coarsen, and replicate various processes of the video-processing and sorting application. The maximum and average times measured over ten repetitions are listed in Table 3.2. The reported numbers confirm the trends observed with the synthetic application. Moreover, if the transformation includes multiple expand or coarsen operations, the measured transformation times are about the same as the sum of the times to perform the individual transformations. The results also show that installing new processes is more costly on the SCC than on the Xeon Phi. This might be due to the framework's ability to load processes dynamically from the file system, which is more costly on the SCC.

In summary, the results demonstrate that transforming an application can take up to several hundreds of milliseconds. However, the time to transform a process strongly depends on the granularity of the refinement network and the work performed per invocation of the FIRE procedure.

3.7.6 A Runtime Scenario

The results presented so far indicate that reasonable speed-ups can be obtained by transforming the application into an alternative network. However, we have also seen that the transformation can take several hundreds of milliseconds. Next, we investigate how the performance of the application is affected during the transformation.

Tab. 3.2: Time in milliseconds to expand, coarsen, and replicate processes of two benchmark applications.

bench-mark	**transformation**	**Xeon Phi**		**SCC**	
		max	**avg**	**max**	**avg**
video-processing	expand v_{dec}	151	78	667	440
	coarsen v_{dec}	155	103	1’054	893
	expand v_{filter}	144	86	740	587
	coarsen v_{filter}	747	428	1’923	1’852
	replicate v_{dec} $5x$	540	461	3’357	2’965
	replicate v_{gauss} $3x$	205	134	5’340	5’260
	replicate v_{sobel} $3x$	213	169	1’961	1’761
	expand v_{filter}, replicate v_{gauss} and v_{sobel} $3x$	855	786	8’017	7’857
	coarsen v_{dec}, expand v_{filter}	275	195	1’423	1’398
sorting	expand v_{sort} $1x$	15	8	115	104
	expand v_{sort} $3x$	73	61	505	470
	coarsen v_{sort} $1x$	93	51	52	42
	coarsen v_{sort} $3x$	596	425	795	675

For this purpose, we measure the frame rate of the video-processing application when the available cores are changed every 40 s, see Fig. 3.10. All resource variations except the one from four to six cores cause a transformation into an alternative network. During the transformation, the frame rate basically stays between the rate at the beginning and end of the transformation. However, it can happen (e.g., when changing the number of available cores from two to five) that several frames arrive at the output process almost at the same time. This happens if multiple replicas start to process simultaneously.

Transforming the application takes the most time when the available cores are increased from six to eight (3.9 s) and reduced from eight to four (4.2 s). In both situations, the runtime manager changes the number of replicas of the ’decoder’, ’gauss’, and ’sobel’ processes. In particular, in the first situation, it replicates the ’decoder’ process three times, changes the number of replicas of the ’gauss’ process from ten to 19, and replicates the ’sobel’ process two times. As the application is assigned the same process network for four and six cores, the runtime-system just reverses the previous transformation when the number of available cores is reduced to four.

Overall, the results show that the proposed transformation technique is able to transform the application into an alternative network seamlessly so that the throughput is never lower than the throughput at the beginning and end of the transformation.

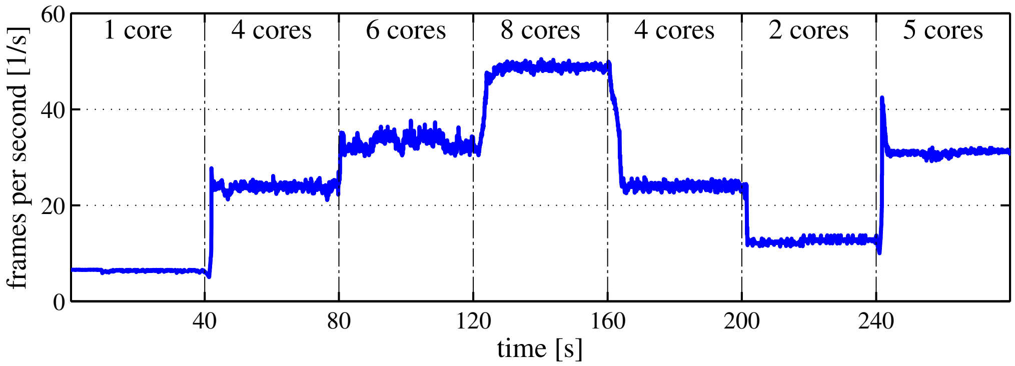


Fig. 3.10: Measured frames per second of the video-processing application when the number of available cores is changed every 40 s.

3.8 Summary

If multiple applications share a heterogeneous many-core SoC, the amount of computing resources available to a single application depends on the other applications that are running on the system and can change over time. This, in turn, complicates the selection of the right degree of application parallelism when specifying the application. Consequently, to exploit the available hardware parallelism efficiently, the application's degree of parallelism must be refined so that inter-process communication and scheduling overheads are minimized.

Tackling this challenge, we have shown that process networks can be specified in a manner that enables the automatic exploration of task, data, and pipeline parallelism. To this end, we have proposed the EPNs semantics, which extends conventional programming models for streaming applications in the sense that several process networks with different degrees of parallelism are abstracted in a single specification. In particular, an application specified as an EPN has a top-level process network defining the initial network. The initial network can be refined by replicating individual processes or replacing stateful processes by other process networks. The latter enables the explicit specification of recursion, commonly used in mathematical and multimedia applications.

To include the proposed concepts in the system design, we extend the DAL design flow by an additional design step, which does not only optimize the mapping, but also the application structure to match the available computing resources. This is particularly useful when the available computing resources are not known at design time, as it is the case when the application is running in multiple execution scenarios and simultaneously with other applications. In this case, the EPN semantics enables the synthesis of multiple design implementations that are all derived from one high-level specification. To adapt the application's degree of parallelism at runtime, we propose a novel technique to transform the application transparently from one process network into an alternative process network without discarding its program state.

Finally, extensive experiments have been carried out on two many-core processors showing the effectiveness of the EPN semantics. Targeting various multimedia streaming applications, we have shown that the execution time of the EPN implementation scales almost linearly with the number of available cores if the proposed extension of the DAL design flow has been used to synthesize multiple implementations of the application automatically. Moreover, we have shown that the proposed transformation technique is able to transform the application seamlessly into an alternative network so that the throughput is never lower than that at the beginning and end of the transformation. This is particularly important if the resources available to a single application can change over time, e.g., when other applications can be started or stopped.

4

Exploiting Multi-Level Hardware Parallelism

4.1 Introduction

As discussed in Section 1.1, future many-core SoCs are predicted to become heterogeneous, thereby coupling general-purpose processors with various hardware accelerators like, for instance, DSPs, Application-Specific Instruction-Set processors (ASIPs), Field Programmable Gate Arrays (FPGAs), or Graphics Processing Units (GPUs). In the context of this chapter, we refer to a processor or a hardware accelerator as a device and consider a SoC to be a set of such devices. Many of these devices are capable to process multiple threads in parallel and some of them are only able to fully exploit their performance if the same instruction is applied to multiple data sources simultaneously.

Exploiting this multi-level parallelism is challenging, in particular if the application specification does not coincide with the available hardware parallelism. For instance, on a GPU, it might be advantageous to split the workload among many small parallel tasks, while on a CPU, it might be more appropriate to use a single task to process the whole workload. A second challenge arises from the programmability of heterogeneous systems. Programming applications for heterogeneous systems can be laborious and costly as different types of compute devices require different code or support different low-level services.

We propose to meet this challenge by running applications on top of the Open Computing Language (OpenCL) framework [Khr10]. OpenCL has been proposed to provide a common interface for programming heterogeneous systems. Nowadays, many hardware vendors, including Intel, AMD, NVIDIA, and STMicroelectronics, provide native support for OpenCL. A program following the OpenCL standard can run on any OpenCL-capable platform that satisfies the resource requirement of the application.

Still, system designers have to manage many low-level details including data exchange, scheduling, or process synchronization, which makes programming heterogeneous systems difficult and error-prone. In fact, previous case studies have shown that OpenCL does indeed provide code portability, but it does not necessarily provide performance portability [WGHP11]. Thus, the use of a higher level programming model to design applications for OpenCL-capable platforms would be desirable as significant areas of the design process could be automated, which would make the complete design process simpler, less error-prone, and more understandable.

Overview

Following these ideas, the contribution of this chapter is the extension of the Distributed Application Layer (DAL) design flow to execute applications specified as process networks on heterogeneous systems using OpenCL. The proposed extension can be considered as a general code synthesis framework to execute process networks on any OpenCL-capable platform.

To exploit the multiple levels of hardware parallelism, a systematic use of pipeline, task, and data parallelism [GTA06, YH09] is proposed. Pipeline and task parallelism are leveraged by distributing the application to the different devices. Pipeline parallelism is achieved by assigning each process of a chain to a different compute device and task parallelism is achieved by executing independent processes on different devices. Finally, data parallelism is used to exploit the parallelism offered by the individual devices: independent process firings are executed simultaneously if the device is able to process multiple threads in parallel and multiple output tokens are calculated in a Single Instruction, Multiple Data (SIMD) fashion [Fly72].

More detailed, we first propose an extension of the previously proposed high-level API for process networks to specify the considered type of data parallelism. To the best of our knowledge, the proposed high-level API is the first API for process networks that enables the system designer to specify the degree of data parallelism flexible in the sense that the best degree of data parallelism can be selected automatically by an optimization framework in accordance with the available hardware parallelism. This extension enables DAL to benefit from target architectures that support SIMD execution.

Afterwards, a runtime-system and program synthesis back-end is presented that enables the efficient distribution of the application among devices from different types and vendors. For instance, the memory location of the FIFO channels is automatically optimized by the runtime-system to improve memory access latencies and end-to-end performance. Seamless integration of input and output operations is provided by the ability to execute processes as native POSIX threads and to transfer tokens between them and processes running on top of OpenCL. Finally, a detailed performance evaluation is carried out to support our claims. In particular, the overhead of the runtime-system is measured, the proposed concepts for exploiting the different levels of parallelism are compared with each other, and the end-to-end throughput is evaluated for various systems.

As an OpenCL program must have fixed-size input and output arrays, only processes reading and writing a fixed number of tokens in every execution of the FIRE procedure are eligible to be executed on top of OpenCL. In fact, processes that show such a characteristics are often called Synchronous Dataflow (SDF) actors [LM87]. However, for readability, we keep our notation and call the parallel entities of a streaming application processes. Processes that do not have a fixed number of input and output tokens are still supported. However, they are executed as native POSIX threads on the CPU so that they cannot benefit from the proposed OpenCL runtime-system.

Outline

The remainder of the chapter is organized as follows: In the next section, related work is reviewed. In Section 4.3, a short overview on OpenCL is given. In Section 4.4, the proposed approach is summarized. In Section 4.5, the previously proposed high-level API for DAL is extended to support the considered type of data parallelism. In Sections 4.6 and 4.7, the proposed OpenCL synthesizer and the runtime-system are discussed. Finally, experimental results are presented in Section 4.8.

4.2 Related Work

We start by reviewing other design flows and high-level frameworks for programming heterogeneous systems, whereby special attention is given to OpenCL-based frameworks.

Maestro [SMV10] is an extension of OpenCL that provides automatic data transfer between host and device as well as task decomposition across multiple hardware devices. While simplifying the task of the programmer tremendously, Maestro introduces new restrictions. For example, tasks have to be independent of each other. The task-level scheduling framework detailed in [SSB+12] extends OpenCL by a task queue enabling a task to be executed on any device in the system. Furthermore, dependencies between tasks are resolved by specifying a list of tasks that have to be completed before a new task is launched. Nonetheless, the burden of organizing data exchange is still left to the designer and no automatic design flow is provided to design applications in a high-level programming language.

Distributed OpenCL (dOpenCL) [KSG12] is an extension of OpenCL to program distributed heterogeneous systems. The approach abstracts the nodes of a distributed system as a single node, but the programmer is still responsible for managing many low-level details of OpenCL.

The high-level compiler described in [DCR+12] generates OpenCL code for applications specified in Lime [ABCR10], a high-level Java compatible language to describe streaming applications. Lime includes a task-based data-flow programming model to express applications at task granularity, similar to process networks. The work in [DCR+12] particularly focuses on generating optimal OpenCL code out of the given Java code. In contrast to the high-level language based approach of Lime, we propose to use a model-based design approach enabling process-to-device mapping optimization and verification. Furthermore, our high-level specification enables the use of a lightweight runtime-system without the need of a virtual machine.

Another approach to design heterogeneous systems is to use a separate programming language for the individual devices. In this context, the execution of process networks on systems with CPUs and GPUs has been studied using NVIDIA's Compute Unified Device Architecture (CUDA) framework [Nvi08] for executing processes on the GPU. For instance, the multi-threaded framework proposed in [BK11a] integrates both POSIX threads and CUDA into a single application. In contrast to our work, the approach is primarily concerned with overlapping communication and computation, but does not optimize the actual memory placement. KPN2GPU, a tool to produce fine-grain data parallel CUDA kernels from a process network specification, is described in [BK11b]. An automatic code synthesis framework taking process networks as input and generating multi-threaded CUDA code is described in [JYH12]. However, they assume that a separate definition of the process is given for the CPU thread implementation and the GPU kernel implementation. In contrast, our API abstracts low-level details enabling the same specification to be used for

CPU and GPU devices. Furthermore, Sponge [HSW$^+$11] is a compiler to generate CUDA code from the StreamIt [TKA02] programming model. Sponge only exploits the coarse-grained task parallelism of the process network while in our approach, the firing of a process is fragmented to fully exploit the multi-level parallelism of today's heterogeneous systems. In addition, our approach generates OpenCL code enabling the same framework to be used for a wider range of heterogeneous platforms than the above described CUDA-targeting frameworks.

4.3 OpenCL Background

OpenCL defines a couple of new terms and concepts, which the reader may be unfamiliar with. For the sake of completeness, a short overview on these shall be given here; for a detailed documentation, however, we refer to [Khr10].

Computational resources are organized hierarchically in OpenCL. There are *devices*, which consist of several *Compute Units (CUs)*; those again are groups of one or more *Processing Elements (PEs)*. As an example, a system with one CPU and one graphics card might provide two OpenCL devices: the CPU and the GPU of the graphics card. The different cores of the CPU would then each be one CU basically consisting of one PE. Each cluster of the GPU would be a CU with typically dozens of PEs. The devices are controlled by the *host*, which is a native program executed on the target machine[1].

The code that runs on the PEs is referred to as *kernels*. Essentially, a kernel is a function written in a C-like programming language called OpenCL C. It should perform a specific task with a well-defined amount of work and then return. All memory portions that a kernel can use to communicate with the other kernels are provided as kernel arguments. When a kernel is executed on a PE, this execution instance is called a *work-item*. When the same kernel is instantiated multiple times at once (e.g., to achieve SIMD execution on GPUs), some of these work-items can be gathered to *work-groups*, which are allowed to share memory for intermediate calculations.

There are two major classes of memory types in OpenCL: global memory and different local memory types. Global memory can be accessed by the host as well as the devices, whereas local memory can only be used by the work-items as an intermediary storage. Note that there is no specification as to where the memory types are physically mapped. In case of a

---

[1] On a personal computer, this means that the CPU may represent the host and a device at the same time; usually, this is implemented by having different operating system threads.

GPU, the global memory would typically reside in the graphics card DDR memory and the local memory in the GPU's fast scratch-pad memories, whereas on a CPU both types just represent the RAM. All memory types have in common that they are limited to the scope of a *context*, i.e., a user-defined set of devices. If, within a context, a global memory reference is passed from one device to another one with a different implementation of global memory, the OpenCL framework will automatically take care of the necessary copying. However, as this is done by the OpenCL back-end, the *driver*, which is implemented by the hardware manufacturers, a context may not contain devices of different manufacturers (e.g., an Intel CPU and a NVIDIA graphics card).

The mapping and scheduling of the work-items are done in a semi-automatic way. The host creates *command queues* for each device and on these queues, it can place commands. A command is either the instantiation of a kernel or a data transfer to or from a global memory. The framework will decide on which PE a work-item is executed and when it will be scheduled. The host can influence this by choosing between *in-order* command queues, which execute the commands strictly in the order in which they were enqueued, and *out-of-order* queues, which may consider later commands if the first command is blocked.

While initially conceived for general purpose GPU programming, OpenCL is supported on many more platforms nowadays. Examples include Intel's Xeon Phi accelerator [Int14], AMD's Accelerated Processing Unit (APU) [AMD13], STMicroelectronics' P2012 platform [BFFM12], and Qualcomm's Snapdragon 805 processor [Qua14a].

4.4 Problem and Approach

In this thesis, a model-based design approach to program heterogeneous systems in a systematic manner is considered. The approach enables the system designer to execute process networks on OpenCL-compatible heterogeneous systems efficiently. The goal is to maximize the end-to-end throughput of an application by leveraging the offered hardware parallelism. Other performance metrics like, for instance, response time, power consumption, or real-time guarantees are not regarded.

Clearly, the goal could be achieved by compiling process networks natively for a specific target platform without the additional layer introduced by OpenCL. Even though the advantages of a model-based design approach could be retained, such an approach would only support a small subset of heterogeneous systems and optimizing the application might be time-consuming. Using OpenCL and its runtime compiler enables the application to take advantage of the latest enhancements of modern processors automatically to maximize the throughput. In comparison with programming applications in OpenCL, the proposed model-based design approach offers a way to leverage various kinds of parallelism explicitly, enables functional verification, and allows to distribute the work efficiently between the different compute devices. Another advantage of the proposed model-based design approach is that the application-independent parts can be implemented once and reused by all applications.

In the following, we first discuss how the proposed design approach exploits the multi-level hardware parallelism in general. Afterwards, we summarize the necessary changes to the DAL design flow to execute applications specified as process networks on heterogeneous systems using OpenCL.

4.4.1 Exploiting the Hardware Parallelism

The key to efficient program execution on heterogeneous systems is to exploit the parallelism they offer. OpenCL supports three levels of hardware parallelism, namely different devices, CUs, and PEs. We do that by leveraging different kinds of application parallelism on each level of hardware parallelism. More specifically, pipeline and task parallelism are used to distribute the process network to the different devices. Afterwards, data parallelism is used to exploit the parallelism offered by the individual devices. Independent process firings are executed concurrently on different CUs of the same device and, to achieve SIMD execution on different PEs, independent output tokens are calculated in parallel. To illustrate how data parallelism is leveraged, we take the process shown in Figure 4.1 as an example. Per firing, it reads six tokens and writes three tokens. Output tokens are independent of each other, although depending on the same input tokens. In our framework, multiple firings might be executed in parallel on different CUs and each output token is calculated on a different PE.

This concept can be implemented in OpenCL by mapping processes, firings, and output tokens onto equivalent OpenCL constructs. The basic idea is that a work-item calculates one or more output tokens of a firing so that all work-items gathered to a single work-group calculate together all output tokens of a firing. All work-items belonging to the same work-group have access to the same input tokens but write their output tokens to different positions in the output stream. This allows SIMD execution.

In addition, multiple firings will be calculated in parallel by having multiple work-groups. Thus, work-items from different work-groups access different input tokens and write their output tokens to different memory positions. Although this could also be achieved by replicating

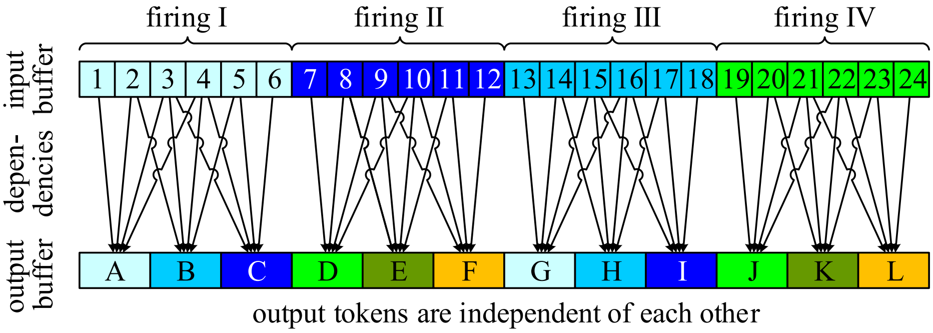


Fig. 4.1: Exemplified process behavior where the calculation of each output token is independent of the other tokens even though three output tokens depend on the same input data.

the process (as described in the previous chapter), the communication and management overhead in this approach is less since multiple invocations can be combined into one invocation. In particular, this has an impact if certain input or output channels connect processes that are mapped onto different devices. The memory required by all firings belonging to the same kernel invocation is then copied between the devices as one block. Note that the number of work-groups per kernel invocation might be limited by the topology of the process network, e.g., if the process is part of a cycle.

When launching a kernel, the OpenCL runtime assigns each work-group to a CU and each work-item to a PE of its work-group's CU. In OpenCL, the number of work-groups and work-items is not bounded by the number of CUs and PEs. In fact, OpenCL can handle more work-groups than available CUs and more work-items than available PEs. The OpenCL device might use them to improve the utilization by switching the context if work-items of a particular work-group are stalled due to memory contention. Figure 4.2 illustrates the different levels of hardware and software parallelism and how they are linked.

4.4.2 Proposed Extension of the DAL Design Flow

Clearly, the hardware parallelism of heterogeneous systems can only be exploited if appropriate design decisions are taken in all steps of the design flow. First, the application specification must be extended with the necessary information about the output tokens. In particular, the challenge is to come up with a high-level API that enables the system designer to specify the functionality of a process independent of the available hardware parallelism so that the proposed application specification does not only provide code portability, but also performance portability. Then,

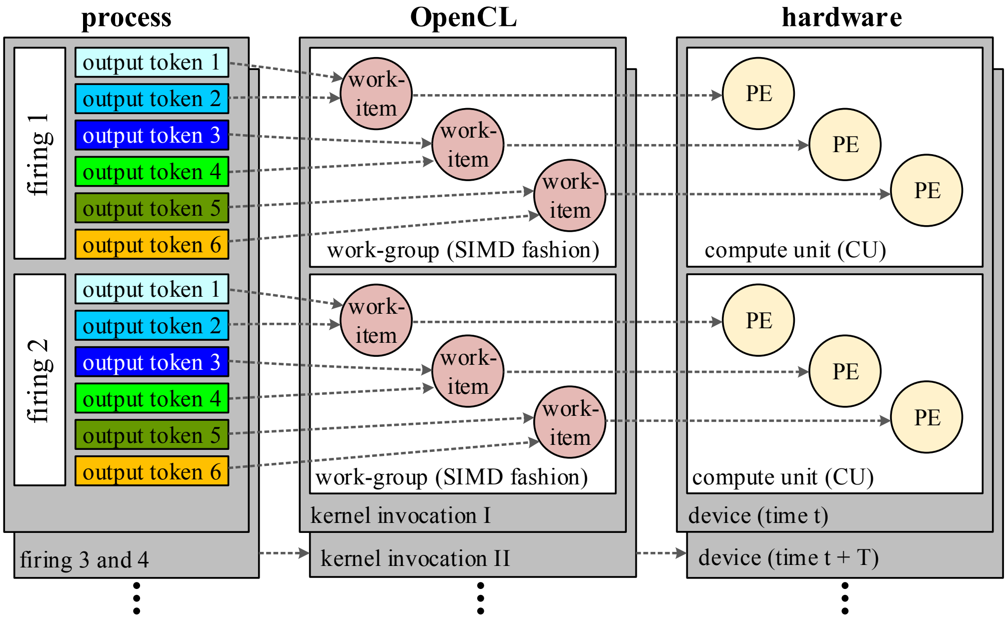


Fig. 4.2: Illustration of the different levels of hardware and software parallelism when executing a process on top of an OpenCL device.

during the design space exploration, decisions must be taken about the number of work-groups and work-items per work-group. The number of work-groups highly depends on the characteristics of the device that the process is going to be executed on. For instance, a loss of performance must be expected if the number of work-groups is not a multiple of the number of CUs. Similarly, the number of work-items might depend on the number of available PEs.

OpenCL C-compliant processes are mapped onto OpenCL devices and the remaining processes (e.g., processes that use file input or output, use recursive functions, or do not read and write a fixed number of tokens in every execution of the FIRE procedure) will be executed as native POSIX threads. During software synthesis, the OpenCL kernel is synthesized based on the previously calculated number of work-groups and work-items and the output tokens are distributed among the work-items.

The runtime-system is in charge of launching the OpenCL kernels at runtime. It does that by forwarding the parallelization directives to the OpenCL framework, which builds and optimizes the kernels so that they leverage the latest enhancements of the device. Furthermore, the runtime-system is in charge of reducing the communication overhead. Efficiently exploiting the hierarchical communication hierarchy of heterogeneous OpenCL platforms is essential for performance, but not trivial as the access to certain memories is restricted to the scope of the context, i.e., a user-defined set of devices. Therefore, the runtime-system must reduce

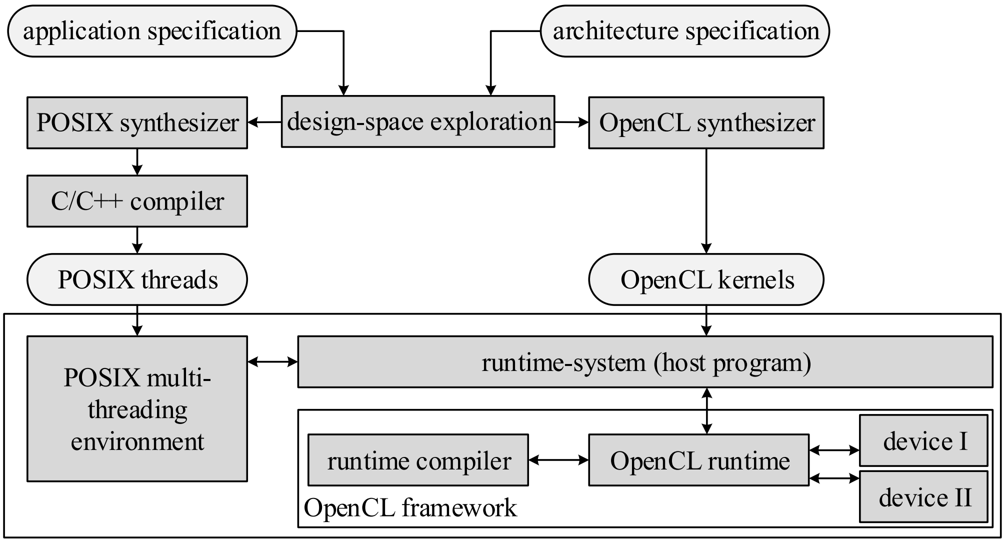


Fig. 4.3: Extension of the DAL design flow to map process networks onto heterogeneous systems using OpenCL. The design flow is essentially composed of five parts, namely the input specification, the design space exploration, the POSIX synthesizer, the OpenCL synthesizer, and the OpenCL runtime-system.

the communication overhead by selecting an ideal memory location for the FIFO channels and combining data transfers. Figure 4.3 summarizes the previously discussed steps.

4.5 System Specification

In this section, we describe the high-level specification we propose for automatic program synthesis. The challenge is to define a high-level API that specifies the degree of data parallelism in a flexible manner so that the best degree of data parallelism can be selected in accordance with the available hardware parallelism during the design space exploration. This enables the system designer to specify the functionality of a process independent of the target platform, which in turn is the key to an application specification that does not only provide code portability, but also performance portability.

The proposed specification is an extension of the high-level API proposed in Section 2.4 that specifies an application as a process network. As previously discussed, only processes that produce and consume a fixed number of tokens per firing can be executed on top of OpenCL. We call this number the *token rate*. In addition, we introduce the notation of *blocks*.

Def. 4.1: **(Block)** *A block is a group of consecutive output tokens that are jointly calculated and do not depend on any other output tokens.*

In the example illustrated in Fig. 4.1, a single output token forms a block as all output tokens are independent of each other. Clearly, all blocks belonging to the same output port have to be of the same size and that size must be a fraction of the port's token rate. This last kind of fragmentation can typically not be leveraged if the application is specified according to an ordinary process network or dataflow graph specification. However, we claim that the proposed extension is natural because, in practice, processes are often specified such that one firing calculates multiple output tokens that require the same input data even though they could be calculated simultaneously. The discrete Fourier and cosine transforms, block-based video processing algorithms [MBM97], and sliding window methods often used for object detection [KT10] are just a few examples of algorithms that exhibit this property.

Based on the above discussion, we extend the XML format shown in Listing 2.2 to specify the token rate and the size of a corresponding block, see Listing 4.1 for an example. The field `rate` specifies the token rate, i.e., the number of tokens produced or consumed per firing, and each output port has a field `blocksize` specifying the size of a corresponding block in number of tokens.

In order to exploit the considered type of data parallelism, the system designer must use a special FOREACH directive to access the individual blocks of an output port. More detailed, the high-level API illustrated in Listing 2.1 has been extended according to Listing 4.2. Providing the system designer the ability to write the code in C/C++ rather than in OpenCL C has the advantage of hiding low-level details of OpenCL. Furthermore, it provides the opportunity to not only execute the process as an OpenCL kernel, but also as a POSIX thread.

Roughly speaking, the INIT procedure is responsible for the initialization and writing the initial tokens to the channels. It is executed once at the startup of the application. Afterwards, the execution of the process is split into individual executions of the FIRE procedure, which is repeatedly invoked by the runtime-system. We leverage the concept of windowed FIFOs [HGT07] for the FIFO channel interface. The basic idea of windowed FIFOs is that processes directly access the FIFO buffer so that expensive memory copy operations are avoided. A window for reading is acquired by using CAPTURE, which returns a pointer to the requested memory. Similarly, a buffer reference for writing is obtained by using RESERVE, potentially together with a block identifier (`blk`, the default is to use the first block). Finally, the FOREACH directive allows the parallel calculation of the individual blocks of an output port. Note that, although

List. 4.1: Specification of a process network with three processes and two channels so that the processes can be synthesized into either OpenCL kernels or POSIX threads. In contrast to the specification shown in Listing 2.2, the token rate and the size of a corresponding block is specified.

```
<processnetwork>
  <process name="producer">
    <port type="output" name="out1" rate="1" blocksize="1"/>
    <source type="c" location="producer.c"/>
  </process>
  <process name="worker">
    <port type="input" name="in1" rate="1"/>
    <port type="output" name="out1" rate="4" blocksize="2"/>
    <source type="c" location="worker.c"/>
  </process>
  <process name="consumer">
    <port type="input" name="in1" rate="2"/>
    <source type="c" location="consumer.c"/>
  </process>

  <channel capacity="8" tokensize="4" name="channel1">
    <sender process="producer" port="out1"/>
    <receiver process="worker" port="in1"/>
  </channel>
  <channel capacity="32" tokensize="1" name="channel2">
    <sender process="worker" port="out1"/>
    <receiver process="consumer" port="in1"/>
  </channel>
</processnetwork>
```

List. 4.2: Implementation of the process "worker" using the proposed API.

```
void INIT(ProcessData *p) {
  initialize();
  FOREACH(blk IN PORT_out1) {
    TOKEN_OUT1_t *wbuf = RESERVE(PORT_out1, blk);
    createinittokens(wbuf, blk); // write initial tokens
  }
}

void FIRE(ProcessData *p) {
  preparation();
  TOKEN_IN1_t *rbuf = CAPTURE(PORT_in1);
  FOREACH(blk IN PORT_out1) {
    TOKEN_OUT1_t *wbuf = RESERVE(PORT_out1, blk);
    manipulate(rbuf, wbuf, blk); // read and write
  }
}
```

not shown here, it is possible to include multiple ports in one FOREACH statement if they all have the same number of blocks per firing. In other words, if a process writes the same number of blocks per firing on each of its output ports, they may be included in the same FOREACH statement.

Finally, the number of work-items per work-group that should be instantiated for a process and the number of work-groups used to gather the work-items is specified as part of the mapping specification. The mapping may also specify a work distribution pattern for every output port. This setting indicates how the output blocks are assigned to the work-items for simultaneous access (consecutive blocks to the same work-item or to different work-items).

4.6 OpenCL Synthesizer

The task of the OpenCL synthesizer is to transform OpenCL-capable processes into OpenCL kernels by performing a source-to-source code transformation. In the following, we illustrate this step based on the "worker" process shown in Listing 4.2.

As described in Section 4.3, an OpenCL kernel specifies the code that runs on one PE. Thus, the basic idea of the OpenCL synthesizer is to replace the high-level communication procedures with OpenCL-specific code so that the FIRE procedure calculates a certain number of output blocks. The number of output blocks depends on the number of work-items per work-group as it is specified by the mapping. When launching the kernel, the runtime-system will specify the number of work-groups and work-items so that one or more firings are executed concurrently by one kernel invocation. Listing 4.3 illustrates how a process is synthesized into an OpenCL kernel:

- The INIT and FIRE procedures are declared as kernels with one parameter per output channel being added to INIT and one parameter per input and per output channel being added to FIRE (Lines 07 and 19 f.).

- Constant helper variables are declared for the token rate and the block size (Lines 02 to 05).

- CAPTURE is replaced with a pointer to the head of the corresponding FIFO channel, using the work-group id to select the correct region (Line 25).

List. 4.3: Embedding the process shown in Listing 4.2 into an OpenCL C kernel. Newly added lines are marked with ■ and modified lines with □.

```
// declare helper variables
const int TOKEN_IN1_RATE = 1;
const int TOKEN_OUT1_RATE = 4, BLOCK_OUT1_SIZE = 2;
const int BLOCK_OUT1_COUNT =
                         TOKEN_OUT1_RATE / BLOCK_OUT1_SIZE;

__kernel void init(__global TOKEN_OUT1_t *out1) {
  int gid = get_group_id(0); // work-group id
  int lid = get_local_id(0); // work-item id
  int lsz = get_local_size(0); // work-item count
  initialization();
  for (int blk=lid; blk<BLOCK_OUT1_COUNT; blk+=lsz) {
    __global TOKEN_OUT1_t *wbuf1 = out1 +
        gid*TOKEN_OUT1_RATE + blk*BLOCK_OUT1_SIZE;
    createinittokens(wbuf, blk); // write initial tokens
  }
}

__kernel void fire(__global TOKEN_IN1_t *in1,
                         __global TOKEN_OUT1_t *out1) {
  int gid = get_group_id(0); // work-group id
  int lid = get_local_id(0); // work-item id
  int lsz = get_local_size(0);  // work-item count
  preparation();
  __global TOKEN_IN1_t *rbuf = in1 + gid*TOKEN_IN1_RATE;
  for (int blk=lid; blk<BLOCK_OUT1_COUNT; blk+=lsz) {
    __global TOKEN_OUT1_t *wbuf1 = out1 +
        gid*TOKEN_OUT1_RATE + blk*BLOCK_OUT1_SIZE;
    manipulate(rbuf, wbuf, blk); // read and write
  }
}
```

- FOREACH is implemented as a loop iterating over the block identifiers (Lines 12 and 26). For each work-item, its ID determines the iteration that it computes. For instance, in Listing 4.3, the solution for a strided work distribution is shown.

  Note that the loop might not be executed by all work-items if the number of work-items is larger than the number of blocks. In practice, this is the case if a process has multiple output ports with each port having a different number of blocks.

- RESERVE is replaced with a pointer to the block being written in the current iteration (Lines 13 f. and 27 f.).

4.7 Runtime-System

The runtime-system's task is to dispatch the OpenCL kernels to the connected devices, which requires two basic functionalities: a framework to synchronize the processes and a memory-aware implementation of the FIFO channels. The implementation of these two functionalities is the key to an efficient execution of process networks on top of the OpenCL framework. In particular, the runtime-system must try to maximize the utilization of OpenCL devices and to reduce communication latencies.

However, maximizing the utilization of an OpenCL device is not trivial. As an OpenCL device can only execute kernels, which should perform a specific task and then return, the runtime-system must ensure that the OpenCL device has always enough executable kernels in its command queue. Another limiting factor for the performance is the memory transfer times if tokens must be transferred between different devices. Here, the challenge is to reduce the number of memory copies and to overlap the computation and communication whenever possible.

To tackle these challenges, we leverage two properties of the OpenCL framework:

In-order command queues. The utilization of a device can be maximized by always having commands in the command queue. In-order command queues allow for an efficient specification of dependencies as they guarantee a fixed execution order of work-items. Unlike all other methods of specifying execution orders, they can be interpreted by the devices autonomously without time-consuming interventions of the host.

Hierarchical memory system. Work-items have to use global memory for data exchange. However, the physical location of these depends on the devices and on the OpenCL directives used by the host. It is therefore important to keep track of these locations and to make sure that the best-suited memory is used whenever possible. The objective is to keep the number of memory copy transactions as low as possible, reducing latencies, and maximizing the end-to-end throughput.

Based on these considerations, we describe the concept of a novel process synchronization framework and the considered communication services in the following.

4.7.1 Process Synchronization

As each kernel invocation executes a specific amount of work and then returns, a mechanism is required to reinvoke the kernel repeatedly with different input data. However, a kernel can only be invoked if enough tokens are available on each input channel and enough space is available on each output channel. Thus, the basic idea of the process synchronization and invocation framework is to monitor the FIFO channels and, depending on their fill level, to invoke the kernels.

Figure 4.4 illustrates the structure of the process synchronization and invocation framework. On start-up, the host creates an in-order command queue for each device and starts the execution-manager. The execution-manager monitors the fill level of the FIFO channels and invokes OpenCL kernels. Callbacks triggered by certain command execution states are used to keep this information up to date. For instance, the completion of a kernel triggers a callback function notifying the execution-manager that a certain amount of tokens has been produced or consumed.

The aim of the execution-manager is to maximize the utilization of the individual devices by avoiding empty command queues. It might even have multiple kernel invocations of the same process simultaneously enqueued, in particular as long as data are available in the input channels and buffer space is available in the output channels. Using in-order command queues is advantageous as the execution-manager can enqueue commands already before the completion of other commands they depend on. For instance, if two processes with a common FIFO channel are mapped onto the same device, the execution-manager can update the number of available tokens of the FIFO channel immediately after enqueuing the source process, even though no tokens have yet been produced.

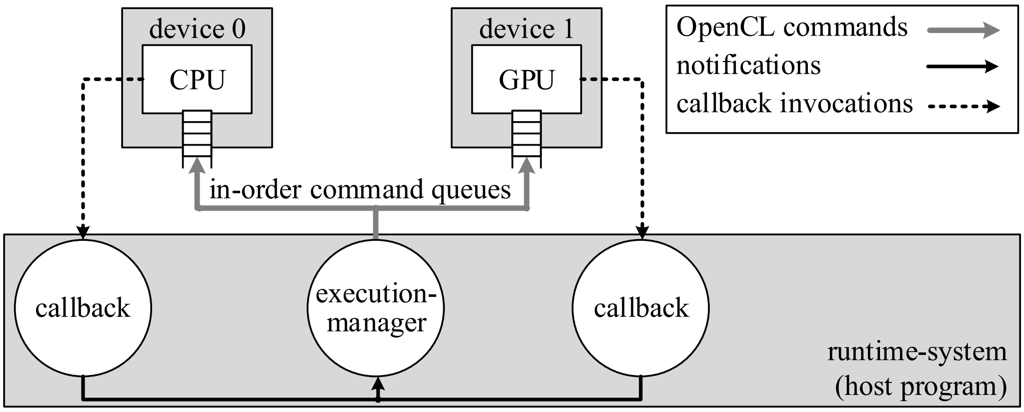


Fig. 4.4: Structure of the process synchronization and invocation framework.

4.7.2 FIFO Communication

The communication service has two tasks to accomplish, namely data transfer and process invocation. While data transfer happens ideally without even involving the host, the host must know the state of the channels, which determines whenever a process can be fired. Therefore, a distributed FIFO channel implementation is considered where the FIFO channel's fill level is managed by the host and only updated by the execution-manager. The memory buffer, on the other hand, may be allocated in the device memory.

In the following, we discuss the FIFO communication between processes mapped onto a single device, between processes mapped onto different devices, and between a process running on top of OpenCL and a process being executed as a POSIX thread.

FIFO Communication on Single Device

If both the source process and the sink process are mapped onto the same device, a buffer is allocated in the global memory of the corresponding device. When firing, the source and sink processes get a pointer to the current tail and to the current head of the virtual ring buffer, respectively. The ring buffer is implemented as a linear buffer in the device memory and OpenCL's sub-buffer functionality is used to split the buffer into individual tokens. Since an in-order command queue is used, the execution-manager can also update the tail and head pointers upon enqueuing the kernels. The FIFO channel implementation and the communication protocol are illustrated in Fig. 4.5.

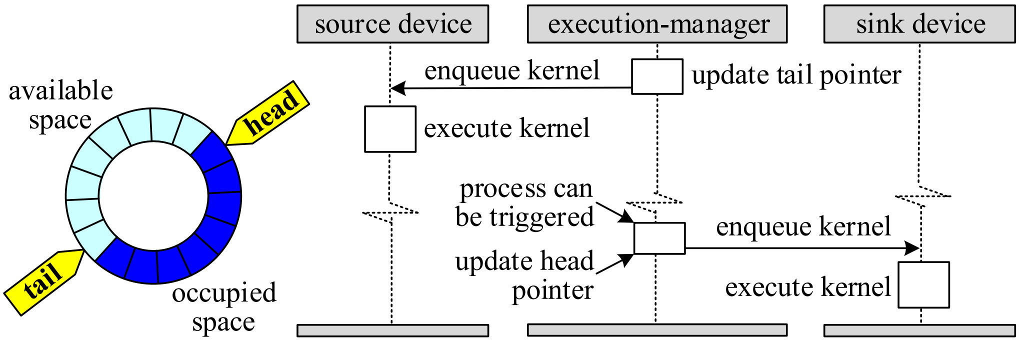


Fig. 4.5: FIFO channel implementation on single devices (left) and corresponding communication protocol (right).

FIFO Communication between Devices

While the FIFO communication implementation on a single device requires no memory copies, it is necessary to transfer data from one device to the other one if two processes mapped onto different devices communicate with each other. In the following, we consider the general case where Host Accessible Memory (HAM) has to be used for this data transfer. The overall idea is to allocate the entire buffer in both global memories of the involved devices and in the HAM. The execution-manager keeps track of all three memories by having three tail and three head pointers. Preallocating the memory is particularly advantageous as expensive memory allocation operations are avoided at runtime.

The communication protocol is illustrated in Fig. 4.6. The underlying principle is that data is forwarded from the source to the sink in packets of reasonable size as soon as possible. The protocol works as follows: After enqueuing the command for launching the source kernel, the execution-manager also enqueues a command for updating the host memory with the newly written data. As in-order command-queues are installed between host and devices, it is ensured that the memory update command is only executed when the kernel execution has been completed. Once the memory update command is completed, a callback tells the execution-manager to update the fill level of the FIFO channel. The execution-manager then checks if the channel contains enough data for a kernel invocation of the sink process. If this is the case, it enqueues a command to update the device memory with the newly available data. Once the sink fulfills all execution conditions, i.e., there are enough tokens and space available on each input and output channel, the execution-manager enqueues the kernel into the command queue, which initiates a new firing of the process. As soon as the kernel execution is completed, a callback tells the execution-manager that the tokens have been consumed.

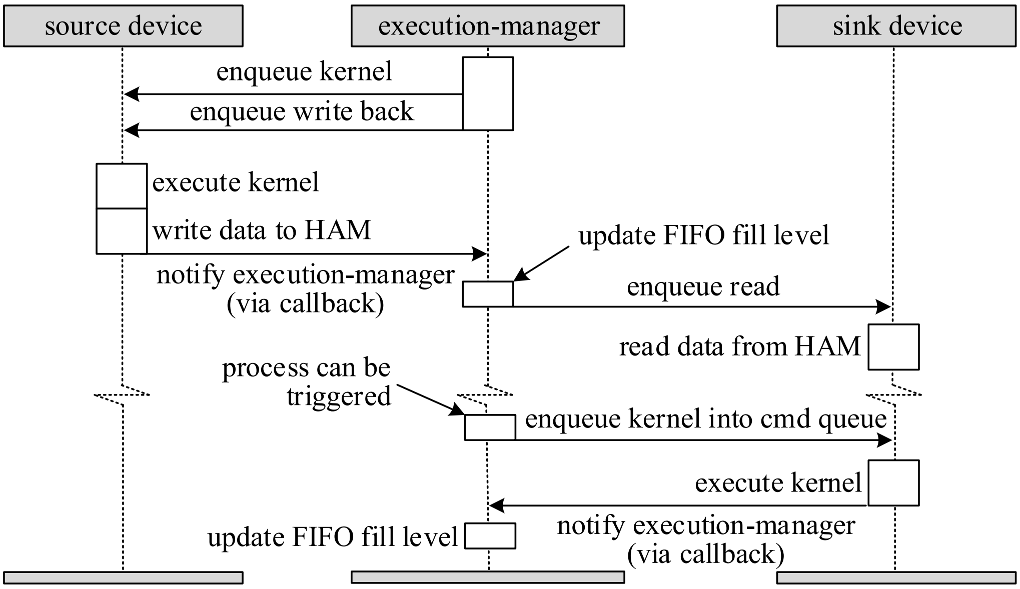


Fig. 4.6: Communication protocol for data exchange between two OpenCL devices.

A special case of this situation is if one of the devices is the CPU. In that case, the global memory of the device is identical with the HAM so that only two buffers are needed.

FIFO Communication between Device and POSIX Thread

The situation that one of the processes is executed as a POSIX thread is similar to the situation where one of the OpenCL kernels is executed on the CPU device. The process executed as a POSIX thread accesses the FIFO buffer in the HAM directly.

4.8 Evaluation

In this section, we evaluate the performance of the proposed code synthesis framework. The goal is to answer the following questions. *a*) What is the overhead introduced by the proposed runtime-system and can we provide guidelines when to execute a process as an OpenCL kernel or as a POSIX thread? *b*) How expensive is communication between processes, when they are mapped onto the same device or onto different devices? *c*) Does the proposed framework offer enough flexibility for the programmer to exploit the parallelism offered by GPUs? — To answer these questions, we evaluate the performance of synthetic and real-world applications on two different heterogeneous platforms.

4.8.1 Experimental Setup

The specifications of the considered target platforms are summarized in Table 4.1. Both CPUs have hyper-threading deactivated and support the Advanced Vector Extensions (AVX) framework. Applications benefit from AVX in the sense that several work-items might be executed in a lock-step via SIMD instructions. The Intel SDK for OpenCL Applications 2013 is used for Intel hardware. OpenCL support for the graphics cards is enabled by the NVIDIA driver 319.17 and by the AMD Catalyst driver 13.1. If not specified otherwise, the used compiler is G++ 4.7.3 with optimization level O2 and default optimizations are enabled in OpenCL.

Tab. 4.1: Platforms used to evaluate the proposed framework.

(a) Evaluation platform A ("desktop computer").

CPU	quad-core Intel Core i7-2600K at 3.4 GHz	
GPU(s)	NVIDIA GeForce GTX 670	AMD Radeon HD 7750
no. of CUs	7	8
no. of PEs per CU	192	64
memory bandwidth	192.2 GB/s	72.0 GB/s
operating system	Arch Linux with kernel 3.7.6-1	

(b) Evaluation platform B ("notebook").

CPU	quad-core Intel Core i7-2720QM at 2.2 GHz
GPU(s)	NVIDIA Quadro 2000M
no. of CUs	4
no. of PEs per CU	192
memory bandwidth	28.8 GB/s
operating system	Ubuntu Linux 12.04 with kernel 3.5.0-28

4.8.2 Overhead of the Runtime-system

First, we quantify the overhead caused by OpenCL and by the proposed process synchronization and invocation mechanism. For this purpose, we have designed a synthetic process network called `producer-consumer` that consists of two processes connected by a FIFO channel. Both processes access the FIFO channel at the same token rate. While the only task of the sink process is to read the received tokens, the source process first performs some calculations before writing the result to the output channel. This result consists of multiple output tokens that can be calculated in parallel using multiple work-items. The number of calculations is

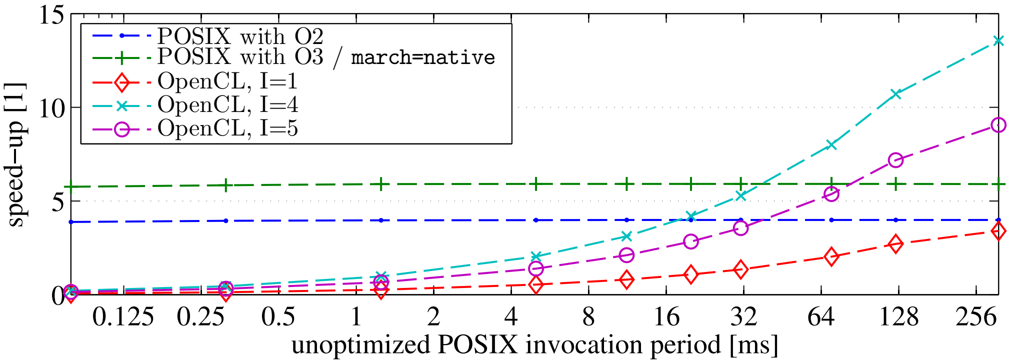


Fig. 4.7: Speed-up of the OpenCL implementations and of the optimized POSIX implementations versus the unoptimized POSIX implementation. *I* refers to the number of work-items.

varied to change the execution time per firing of the source process. This also changes the invocation period and its inverse, i.e., the invocation frequency as the process network's invocation interval is only limited by the execution time of the source process. For brevity, we only report the results for platform A. However, the results for platform B exhibit similar trends.

Overhead of the OpenCL Framework

To quantify the overhead of the OpenCL framework, we have synthesized both processes of the `producer-consumer` application for either OpenCL or POSIX. When synthesizing the application for POSIX, either no optimization, optimization level O2, or optimization level O3 with the setting `march=native` is used. If `march=native` is set, G++ automatically optimizes the code for the local architecture. When synthesizing the application for OpenCL, the number of work-groups is set to one so that each process is executed on exactly one core.

Figure 4.7 shows the speed-up of the OpenCL implementations and of the optimized POSIX implementations versus the execution time of the unoptimized POSIX implementation, with the number of calculations in the source process being varied. The x-axis represents the invocation period of the unoptimized POSIX implementation. As the kernel invocation overhead in OpenCL is nearly independent of the kernel's amount of work, the POSIX implementation performs better for small invocation periods. On the other hand, the overhead is less crucial for longer invocation periods and OpenCL implementations achieve even higher speed-ups than the optimized POSIX implementations. This may be due to OpenCL's ability to utilize the CPU's vector extension so that four

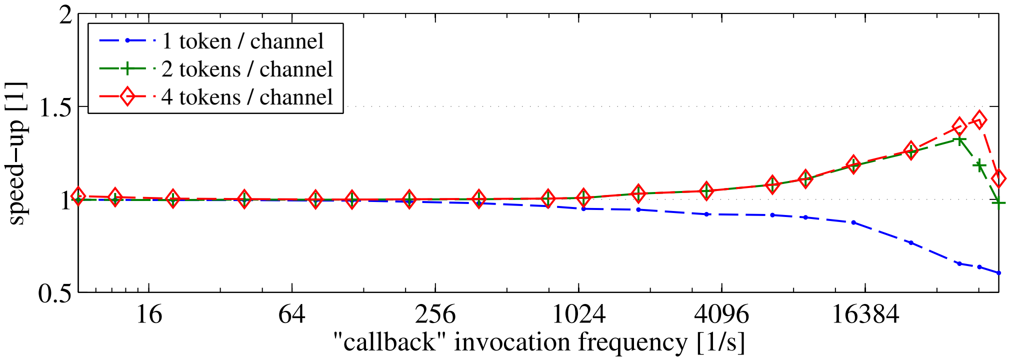


Fig. 4.8: Speed-up of the `producer-consumer` application when using the "manager" mechanism relative to the execution time when using the "callback" mechanism.

work-items are executed in SIMD fashion. That assumption is supported by the fact that the used CPU is only able to execute four work-items in parallel. Therefore, distributing the work to five work-items is counterproductive. Note that G++ also makes use of the AVX commands when the corresponding option is enabled. This explains why the speed-up of the O3 implementation is always higher than the speed-up of the OpenCL implementation with one work-item.

Overhead of the Runtime-System

To evaluate the overhead caused by the proposed runtime-system, we synthesize the `producer-consumer` application for OpenCL with one work-item per process and measure its execution time for two different process synchronization mechanisms. The first mechanism is called "manager" and corresponds to the proposed process synchronization mechanism presented in Section 4.7, i.e., an execution-manager monitors the FIFO channels and creates new kernel instances. A larger channel capacity allows the execution-manager to have multiple firings of the same process enqueued in the command queue simultaneously, which can lead to a higher utilization of the device. The second mechanism is called "callback" and is a naive approach that uses the callback functions of OpenCL (e.g., kernel execution finished, data transfer completed) for enqueueing further commands.

Figure 4.8 shows the speed-up of the `producer-consumer` application when using the "manager" mechanism relative to the execution time when using the "callback" mechanism. Both processes are mapped onto the CPU and the number of calculations in the source process is varied. The x-axis represents the invocation frequency when using the "callback" mechanism that, as previously described, is inversely proportional to the number of calculations in the source process. As expected, no speed-up is achieved for low invocation frequencies. For higher invocation frequencies, the "manager" mechanism is slower than the "callback" mechanism if the FIFO channel has a capacity of one token. In this case, both mechanisms can enqueue a new firing only if the previous firing is completed. However, the feedback loop is larger for the "manager" mechanism as it also includes the execution-manager. If the FIFO channel has a capacity of more than one token, the "manager" mechanism can enqueue a new firing in parallel to the execution of the old one so that the "manager" mechanism is faster than the "callback" mechanism. Finally, for very large invocation frequencies, the firing completes earlier than the "manager" mechanism can enqueue a new firing so that the speed-up declines again.

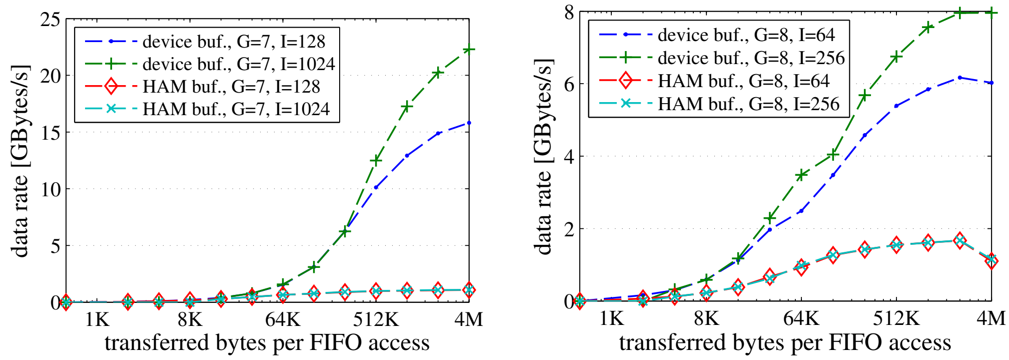
Overall, the results demonstrate that the proposed execution-manager performs considerably better than a naive process invocation mechanism performs. Furthermore, we claim that OpenCL is not only useful for executing processes on GPUs, but also on CPUs if data parallelism can be leveraged efficiently.

4.8.3 Intra- and Inter-Device Communication

Next, we evaluate the communication costs for different types of FIFO channel implementations. The goal is to show that mapping the memory buffers onto the distributed memory architecture in a sub-optimal manner may affect the overall performance of the application.

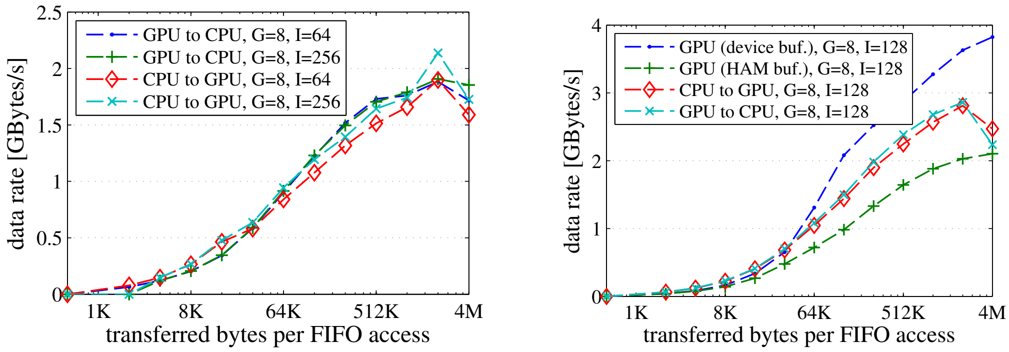
For this purpose, we measure the data transfer rate between two processes mapped onto either the same device or different devices. Our test application is designed such that in each firing, the 'producer' process generates one token, which it fills with a simplistic integer sequence. Then, the 'producer' process sends the token to the 'consumer' process that reads the values. This set-up is selected to ensure that the workload per transmitted byte is independent of the size of a token. Figure 4.9 shows the aggregated data rate for different process mappings and channel implementations, where the size of a single token is varied between 512 bytes and 4 MBytes. The channel size is fixed to 32 MBytes.

With the first set-up, the data transfer rate between two processes mapped onto the same device is measured for two different FIFO implementations. The results are shown in Figs. 4.9a and 4.9b for the NVIDIA and the AMD CPU, respectively. First, we use the optimized FIFO implementation, which keeps the data in the global memory of the device ("global buf."). Second, we used a naive FIFO implementation, which transfers data through HAM ("HAM buf."). The number of work-groups (G) per process is set to the number of CUs and the number of work-items



(a) Platform A, NVIDIA GPU.

(b) Platform A, AMD GPU.



(c) Platform A, AMD GPU and CPU.

(d) Platform B.

Fig. 4.9: Data rate for different process mappings and channel implementations. The number of work-groups (G) per process is set to the number of CUs and the number of work-items per work-group is indicated by *I*.

per work-group is indicated by *I*. Having more work-items might lead to higher data transfer rates as more PEs can read and write concurrently. The observed peak data rates are 20.30 GBytes/s when both processes are mapped onto the NVIDIA GPU and 7.96 GBytes/s when both processes are mapped onto the AMD GPU. The data transfer rate is considerably lower if the memory buffer is allocated in the HAM. In this case, the observed peak data rates are 1.09 GBytes/s and 1.67 GBytes/s when both processes are mapped onto the NVIDIA GPU and both processes are mapped onto the AMD GPU, respectively.

The data transfer rate for the case that one process is mapped onto the CPU and the other process is mapped onto the AMD GPU is illustrated in Fig. 4.9c. "GPU to CPU" means that the 'producer' process is mapped onto the GPU and the 'consumer' process is mapped onto the CPU. For "CPU to GPU", it is vice versa. The observed peak data rates are 1.91 GBytes/s and 2.14 GBytes/s when the 'producer' process is mapped onto the GPU and the CPU, respectively.

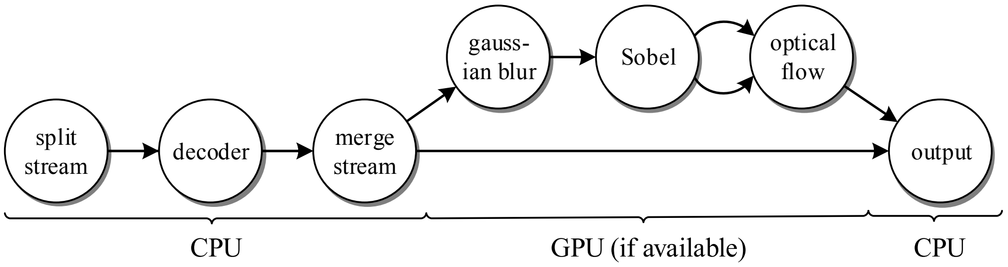


Fig. 4.10: Process network of the video-processing application.

Finally, Fig. 4.9d shows a summary of the data transfer rates for platform *B*. The observed peak data rate is 3.82 GBytes/s and measured when both processes are mapped onto the GPU.

We conclude that exploiting on-device communication without going through HAM is essential for the performance. Yet, we think that the communication costs still leave much room for improvement. For instance, a higher data rate might be achieved by using the local memory of a CU and by overlapping local to global memory communication with computation.

4.8.4 Exploiting Data and Task Parallelism

The results presented so far indicate that the proposed concepts of multilevel parallelism indeed lead to higher performance. Next, we will investigate this question further by comparing the performance of a real-world application for different mappings and degrees of parallelism.

For this purpose, a video-processing application has been implemented that decodes an MJPEG video stream and then applies a motion detection method to the decoded video stream, see Fig. 4.10 for the process network. The MJPEG decoder can decode multiple video frames in parallel but cannot divide the output tokens into smaller pieces. The motion detection method is composed of a Gaussian blur, a gradient magnitude calculation using Sobel filters, and an optical flow motion analysis. Tokens transmitted between these three components correspond to single video frames, but in all filters, the calculation of an output pixel is independent of the other output pixels. A gray scale video of 320 × 240 pixels is decoded and analyzed in all evaluations. If a GPU is available, the 'Gaussian blur', the 'Sobel', and the 'optical flow' processes will be mapped onto it. Otherwise, all processes will be mapped onto the CPU.

We measure the frame rate of the application for different degrees of parallelism and different target platform configurations. The number of work-groups of the 'decode' process is fixed to three and the number of work-groups of the 'split stream', 'merge stream', and 'output' processes are fixed to one. Furthermore, the number of work-groups (G) and the number of work-items (I) per work-group are varied for the 'Gaussian blur', the 'Sobel', and the 'optical flow' processes. However, for the sake of simplicity, all three processes have the same number of work-groups and work-items.

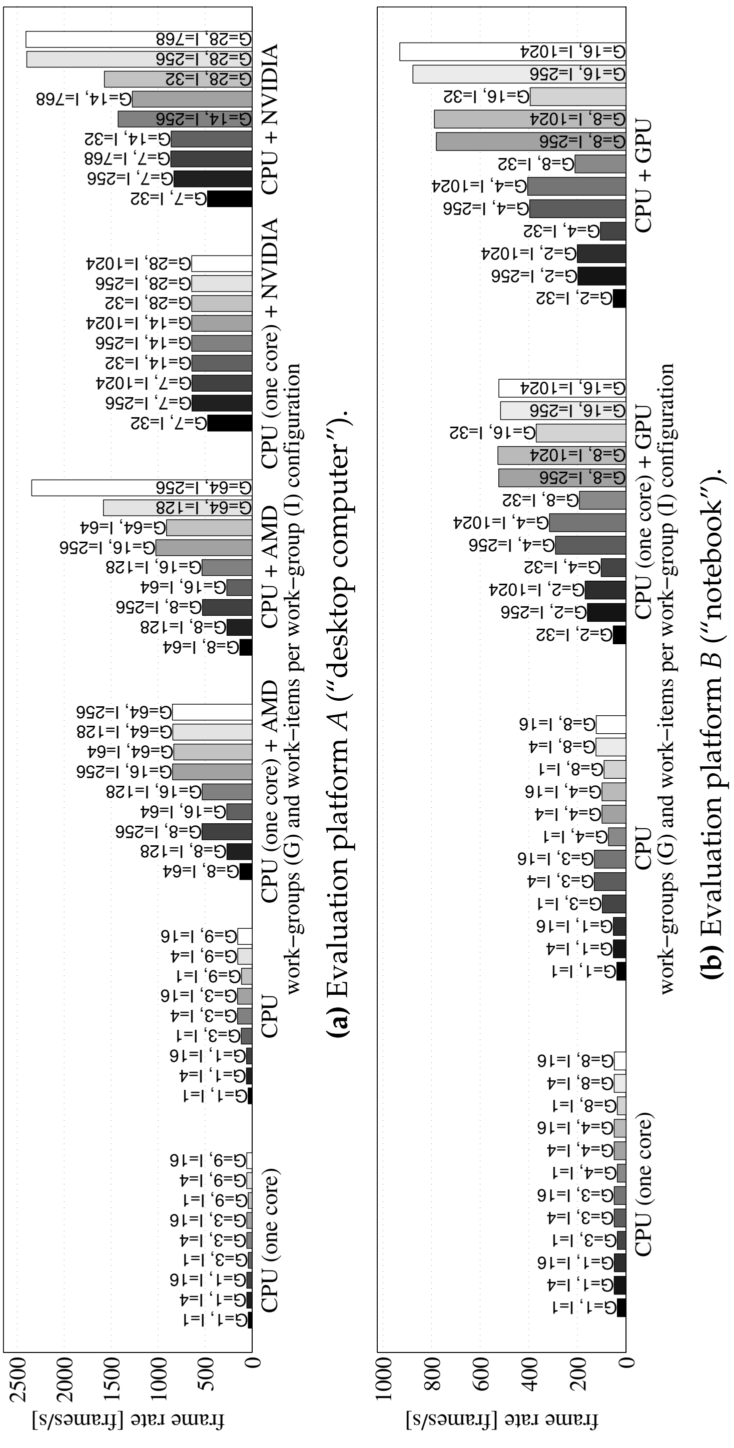
Figure 4.11a shows the frame rates achieved with different configurations on target platform *A*. The highest performance (2'347 frames/s) is achieved with all CPU cores and a GPU device being available. In that case, the bottleneck is not anymore the GPU, but the CPU that is not able to decode more frames. Similarly, the CPU is mainly the bottleneck if only one core is used to decode the frames. Mapping all processes onto a single core of the CPU leads to a maximum frame rate of 57 frames/s. Thus, compared to the case where all processes are mapped onto one CPU core, a speed-up of almost $41x$ is achieved when all CPU cores are used together with a GPU device. The plot also shows that the GPU can highly benefit from a large number of work-items. Finally, note that a different work distribution pattern is used for the calculation of the individual output pixels depending on whether the 'Gaussian blur', the 'Sobel', and the 'optical flow' processes are mapped onto the CPU or the GPU. Mapping consecutive blocks to the same work-item works best for the CPU while consecutive blocks to different work-items works best for the GPU.

Figure 4.11b shows the frame rates for different configurations on target platform *B*. Again, the peak performance (931 frames/s) is achieved when all cores of the CPU and the GPU device are available. It constitutes a speed-up of $19x$ compared to the case where all processes are mapped onto one CPU core. The plot also shows that the number of work-groups should be aligned with the available hardware. We have found that the Intel OpenCL SDK version we used distributes the OpenCL kernels to only three cores, which is why a higher frame rate is obtained when executing three work-groups instead of four.

Overall, the results demonstrate that the proposed framework provides developers with the opportunity to exploit the parallelism provided by state-of-the-art GPU and CPU systems. In particular, speed-ups of up to $41x$ could be measured when outsourcing computation intensive code to the GPU.

4.9 Summary

Many-core SoCs integrate a wide variety of processors including general-purpose processors, GPUs, or accelerators. Many of these processors are able to process multiple threads in parallel and some of them are only able to fully exploit their computing power if the same instruction is



(a) Evaluation platform *A* ("desktop computer").

(b) Evaluation platform *B* ("notebook").

Fig. 4.11: Frame rate of the process network outlined in Fig. 4.10 for different degrees of parallelism and different target platform configurations.

concurrently applied to multiple data sources. However, exploiting this multi-level parallelism is challenging.

Yet, another challenge of heterogeneous systems is their programmability. OpenCL is a first attempt to provide a common interface for all components of a heterogeneous system. A program following the OpenCL standard can run on any OpenCL-capable platform that satisfies the resource requirements of the program. Still, system designers have to manage many low-level details including data exchange, scheduling, or process synchronization, which makes programming heterogeneous systems difficult and error-prone.

Thus, it would be desirable to have a higher level programming interface to design heterogeneous many-core SoCs as significant areas of the design process could be automated, making the complete process simpler, less error-prone, and more understandable. Tackling this issue, the contribution of this chapter is an extension of the DAL design flow to execute process networks efficiently on any heterogeneous OpenCL-capable platform.

The outcomes of this chapter are twofold. First, a systematic approach to exploit the multi-level parallelism of a heterogeneous system is proposed. Task and pipeline parallelism are used by the proposed approach to distribute an application among the different compute devices. Afterwards, the parallelism offered by the individual compute devices is exploited by means of data parallelism. In other words, multiple firings of a process are concurrently processed and independent output tokens are calculated in SIMD mode. Second, a general code synthesis framework to execute DAL applications on any OpenCL-capable platform is proposed. In this framework, processes are automatically embedded into OpenCL kernels and scheduled by a centralized execution-manager. FIFO channels are instantiated by the runtime-system in a way that memory access latencies and end-to-end performance are improved.

We demonstrated the viability of our approach by running synthetic and real-world applications on two heterogeneous systems consisting of a multi-core CPU and multiple GPUs achieving speed-ups of up to $41x$ compared to execution on a single CPU core. Furthermore, we illustrated that OpenCL is not only useful for executing processes on GPUs, but also on CPUs if data parallelism can be leveraged efficiently.

Even though the discussed approach is restricted to processes reading and writing a fixed number of tokens in every execution of the FIRE procedure, extending it to a more general process model is possible. For instance, the approach can be extended to the case where the system only knows the amount of tokens that it should read in the next execution of the FIRE procedure. Furthermore, it is enough to know an upper bound on the number of tokens written in one execution of the FIRE procedure. Finally, we note that the extension of the framework to distributed systems with multiple host controllers is straightforward, e.g., by connecting the different nodes by the Message Passing Interface (MPI) communication protocol, as discussed in Section 2.6 in the context of Intel's Single-chip Cloud Computer (SCC) processor.

5

Thermal-Aware System Design

5.1 Introduction

The use of deep submicrometer process technology to fabricate SoCs has imposed a major rise in power densities, which in turn threatens the reliability and performance of SoCs [HFFA11]. The induced high chip temperatures may cause long-term reliability concerns and short-term functional errors. To make matters worse, recent studies have predicted that three-dimensional stacking [Loh08] will magnify the problem of high chip temperatures in the near future [ZGS+08].

To reduce device failures, the cooling system has to be designed for the worst-case chip temperature, i.e., the maximum chip temperature under all feasible scenarios of task arrivals, which can lead to a waste of energy due to over-provisioning [CRWG08]. Besides improving the cooling system, thermal and reliability issues have been tackled by reactive thermal management techniques or thermal-aware task allocation and scheduling algorithms. In particular, reactive thermal management techniques such as Dynamic Voltage and Frequency Scaling (DVFS) keep the maximum temperature under a given threshold [DM06, IBC+06] by stalling or slowing down the processor. However, the drawbacks of reactive thermal management techniques are the unpredictable runtime overhead and the unexpected performance degradation [BJ08, CRWG08], with the consequence that latencies and other performance metrics depend on the temperature. Therefore, providing guarantees on the maximum temperature is as important as functional correctness and timeliness when designing embedded real-time many-core SoCs.

When the workload of the system is known, pro-active thermal-aware allocation and scheduling techniques that avoid thermal emergencies, and thus a reduction in performance, might be preferable over reactive thermal management techniques. By selecting an optimal frequency, voltage, and task assignment, the peak temperature can be reduced significantly so that latency requirements and other performance metrics can be guaranteed at design time, independent of the temperature [MMA+07, CDH08, CRWG08, FCWT09]. To this end, prior work either lowered the average temperature or assumed deterministic workload where the maximum temperature of the system can be calculated by simulating the system. However, unknown input patterns cause the workload to be non-deterministic so that the maximum possible chip temperature under all feasible scenarios of task arrivals is difficult to identify. Only when the corner case that actually leads to the maximum temperature of the system is considered, simulation-based thermal analysis techniques do not lead to an undesired underestimation of the maximum temperature.

Ex. 5.1: *We illustrate the problem of giving guarantees on the maximum temperature of a MPSoC or many-core SoC by means of a simple example. We assume a system with three homogeneous work-conserving processing cores*[1]*, two of them processing a periodic event stream with jitter, see Table 5.1 for the parameters of the computational model. Whenever they process some events, the cores are in 'active' mode and consume both dynamic and leakage power. Otherwise, they are in 'idle' mode and consume only leakage power, see Section 5.8 for a summary of the considered hardware model. Since the second processing core has no workload assigned, it only consumes leakage power. Nonetheless, when another core is processing, the temperature of the second core increases, as well, due to the heat flow between neighboring components.*

Now, we compare various methods to calculate the maximum temperature:

- *By means of the average utilization of the cores, we can calculate the average power consumption and find a maximum steady-state temperature of* 345.1 K.
- *We find* 351.3 K *for the maximum temperature, when executing* 40 *traces with random jitter on a thermal-aware cycle-accurate simulator consisting of the MPARM virtual platform [BBB+05] and HotSpot [HGV+06].*
- *Using the method developed in this chapter, we get a tight worst-case chip temperature of* 352.0 K.

---

[1] A work-conserving core has to start processing as soon as there is an event in its ready queue.

Tab. 5.1: Parameters of the computational model for the motivating example.

	period	**jitter**	**computing time**
processing core 1	240 ms	480 ms	120 ms
processing core 2	*no workload*		
processing core 3	120 ms	240 ms	60 ms

Figure 5.1 outlines the temperature evolution of core 1 *when the system is processing the thermal critical workload trace leading to the worst-case chip temperature together with the temperature evolution of* 40 *traces with random jitter executed on the thermal-aware cycle-accurate simulator. As shown, none of the described established methods for calculating the maximum temperature is able to give guarantees on the worst-case chip temperature of this system.*

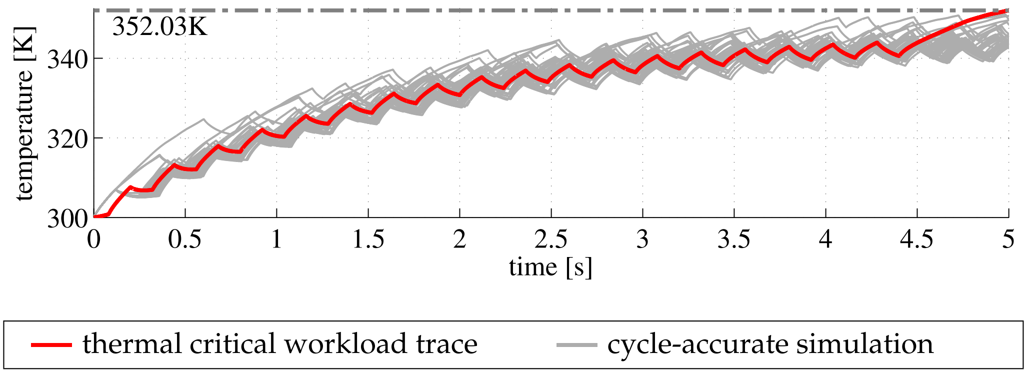


Fig. 5.1: Temperature evolution of processing core 1 processing the example system described in Table 5.1.

Overview

In this chapter, we extend the Distributed Application Layer (DAL) design flow introduced in Chapter 2 with a high-level optimization framework for mapping real-time applications onto embedded many-core SoCs that provides guarantees on both temporal and thermal correctness. More specifically, the framework aims at either ruling out mapping alternatives that do not conform to real-time and peak temperature requirements or finding the mapping alternative that reduces the worst-case chip temperature the most. The temporal and thermal characteristics of the mapping candidates are compared by means of formal worst-case real-time analysis methods so that safe bounds on the execution time and the maximum chip temperature can be provided.

More detailed, we first present an analytic method to calculate an upper bound on the maximum temperature of a many-core SoC with nondeterministic workload. The considered thermal model is able to address various thermal effects like heat exchange between neighboring cores and temperature-dependent leakage power. We use the well-established standard event model [HHJ+05] to model non-determinism in the workload, i.e., we consider periodic event streams with jitter and delay. Real-time calculus [TCN00], a formal method for schedulability and performance analysis of real-time systems, is applied to upper bound the workload that might arrive in any time interval. Our method then identifies the critical workload trace that leads to the worst-case chip temperature. The only requirement of the method is that the real-time scheduling algorithms are work-conserving. However, this applies to most of the traditional scheduling algorithms like, for example, Earliest-Deadline-First (EDF) [XP90], Rate-Monotonic (RM) [LL73], Fixed-Priority (FP) [Ser72], or Deadline-Monotonic (DM) [ABRW91].

Then, we integrate the proposed thermal analysis method into a design-space exploration framework intended to optimize the process-to-core assignment of a many-core SoC. To this end, we implement the proposed thermal analysis method as an extension of the Modular Performance Analysis (MPA) framework [WTVL06] so that thermal and temporal analysis can be done within the same framework. To explore the design space without user-interactions, the analysis models are automatically generated from the same set of specifications as used for software synthesis. Moreover, to increase the model accuracy, the analysis models are calibrated with data corresponding to the target platform. The data is obtained in an automatic manner prior to design space exploration by either simulation on a virtual platform or execution on the real hardware. Finally, we formulate the optimization problem with the objective to minimize the worst-case chip temperature and solve the optimization problem using simulated annealing [KGV83].

Outline

The remainder of the chapter is organized as follows: First, related work is discussed in the next section. In Section 5.3, the mapping optimization framework is presented. In Section 5.4, the computational and thermal models used for formal system analysis are introduced. The thermal analysis method is described in Section 5.5. The automated generation and calibration of the thermal analysis models is presented in Section 5.6. In Section 5.7, the assignment of processes to cores is formulated as an optimization problem. Finally, case studies to highlight the viability of our methods are presented in Section 5.8.

5.2 Related Work

Nowadays, reactive thermal management techniques [BM01, DM06, KSPJ06] are typically used to address thermal issues in general-purpose computing systems. For example, multiple architectural-level techniques for thermal management like DVFS and stop-go scheduling are evaluated in [DM06]. However, the use of reactive thermal management techniques often causes a significant degradation of performance [BJ08] or leads to expensive runtime overhead and high complexity [CRWG08]. Therefore, the use of reactive thermal management techniques is often undesirable in today's embedded systems, in particular when tackling real-time constraints. On top of that, reactive thermal management techniques become less effective as the operation voltage is limited by saturation [WDW10]. The alternative is to adopt system-level mechanisms and address thermal issues at design time.

Thermal-aware task allocation and scheduling algorithms for MPSoCs are explored in [DM06, XH06, CRW07, MMA+07, CDH08, CRWG08, FCWT09]. For instance, thermal-aware heuristics to reduce the maximum and average temperature are compared with power-aware heuristics to reduce the maximum power consumption in [XH06]. Thermal management techniques for systems with unknown workload, like load balancing or temperature aware random scheduling, are discussed in [CRW07]. The thermal-aware scheduling problem is formulated as a convex optimization problem in [MMA+07] and a mixed-integer linear programming formulation to reduce the peak temperature of MPSoCs is proposed in [CDH08]. In [CRWG08], a similar problem is solved to minimize the energy consumption and to reduce thermal hot spots. Finally, a global scheduling algorithm to reduce the peak temperature while running all cores at their preferred speed is proposed in [FCWT09].

All these design time methods have in common that the temperature analysis is performed by either simulation or steady-state analysis. More detailed, evaluating the temperature characteristics is typically a two-step process. First, the transient power dissipation of the system is determined by means of a power-aware simulator, either software-based [BCT+10] or hardware-based [GdVA10]. Afterwards, the measured power dissipation is used to either calculate the average temperature based on a steady-state analysis [YCTK10] or evaluate the transient temperature evolution in a thermal simulator. HotSpot [SSS+04, HGV+06] and 3D-ICE [SRV+10] are the most common examples of such thermal simulators. However, due to the complexity of today's systems, it is difficult to identify corner cases that actually lead to the maximum temperature of the system under all feasible execution traces. Consequently, simulation-based thermal analysis methods may lead to an undesired underestimation of the maximum temperature as illustrated in Example 5.1.

In this work, we use a different approach. Similar to well-known best-case and worst-case timing analysis methods for multiprocessor systems [HHJ+05, WTVL06], we use formal analysis methods to predict the maximum temperature of a real-time system. A first attempt to calculate the worst-case chip temperature of an embedded system has been proposed in [RYB+11]. However, as it does not incorporate the heat transfer among neighboring cores, the method cannot be applied to multi- and many-core systems.

5.3 Mapping Optimization Framework

The aim of this chapter is to extend the DAL design flow with a high-level optimization framework that maps a process network onto a many-core SoC platform in a time and thermal optimal manner. To this end, the proposed framework considers the optimization criteria worst-case performance and worst-case chip temperature.

Presenting a framework for all kinds of applications and architectures that can be specified in the context of the DAL framework is beyond the scope of this chapter. In fact, this chapter focuses on performance analysis using MPA [WTVL06], which restricts the class of process networks that can be analyzed automatically. In the following, we first discuss the application, architecture, and mapping models considered in this chapter. Afterwards, we summarize the proposed high-level optimization framework. We will use the example system shown in Fig. 5.2 to illustrate the different models and notations.

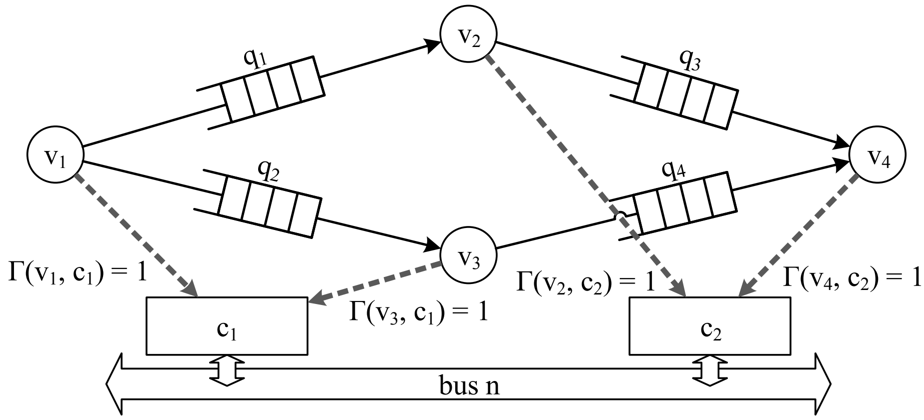


Fig. 5.2: Example system to illustrate the considered system specification.

5.3.1 Application Model

In this chapter, we consider process networks that can be specified as dataflow process networks [LP95] and whose processes have a workload that is bounded in any time interval $\Delta \geq 0$. Dataflow process networks are a special case of Kahn Process Networks (KPNs) that execute their FIRE procedures only if enough input data is available. This has the advantage that the context switching overhead caused by the blocking read semantics of KPNs is avoided.

These restrictions allow us to analyze the system with MPA that abstracts the workload of every independent component by a so-called arrival curve. However, even though these restrictions already limit the set of applications that can be analyzed by the proposed mapping optimization framework, the following assumptions first stated in [Hai10] have to be made in order to avoid a mismatch between high-level analysis model and actual system behavior:

- In each execution of the FIRE procedure, a process first reads tokens from its input streams, processes them, and finally writes tokens to the output streams.
- The process network does not contain cycles and the size of the FIFO channels are selected so that the processes are never blocked when writing.
- The worst-case execution time of a process is independent from the execution of other processes mapped onto the same core.
- The time that a process needs to write to and read from a FIFO channel is constant apart from variations due to varying token sizes.

The interested reader is referred to [Hai10] for more details on how these assumptions reduce the gap between system behavior and analysis model. Next, we formally define a dataflow process network.

Def. 5.1: **(Dataflow Process Network)** *A dataflow process network $p = \langle V, Q \rangle$ is represented as a directed, connected graph where each node in the graph represents a process $v \in V$. Edges represent unbounded FIFO channels $q \in Q$, which in turn are used by processes to communicate with each other. A process $v \in V$ is blocked as long as not all of its predecessor processes have produced the number of tokens that the process needs to be triggered.*

Abstractly, a process can be modeled as a stream of events. We suppose that an event has to complete its execution within D_v time units after its arrival. The separation between processes for computation and channels for communication enables the modular generation and calibration of analysis models from the same specification as used for actual system synthesis. Note that processes are commonly referred to as actors in the context of dataflow process networks. However, for readability, we again keep our notation and call the computation entities of a streaming application processes.

5.3.2 Architecture Model

Similar to Chapter 3, we model a heterogeneous many-core architecture $\mathcal{A} = \langle C, n \rangle$ as a set of cores C that are connected by a communication network n, e.g., a bus or a NoC. However, unlike temporal analysis, thermal analysis requires a detailed description of the architecture to model the heat flow between neighboring cores. This is achieved by representing the placement of the cores by a floorplan. See [ADVP+07] for various examples of SoC floorplans.

5.3.3 Mapping Model

The mapping specification describes both the binding Γ of processes to processing cores and their scheduling σ on shared resources.

Def. 5.2: **(Binding Function)** *The binding of a dataflow process network* $p = \langle V, Q \rangle$ *onto a many-core architecture* $\mathcal{A} = \langle C, n \rangle$ *is specified by the function* $\Gamma(v, c)$ *that is* 1 *if a process* v *is assigned to core* c *and* 0 *otherwise:*

$$\Gamma(v,c) = \begin{cases} 1 & \textit{if process } v \textit{ executes on core } c, \\ 0 & \textit{otherwise.} \end{cases} \tag{5.1}$$

The scheduling policy is supposed to be work-conserving, i.e., the processing component has to process as soon as there is data available. This assumption applies to most traditional scheduling algorithms like, for example, EDF, RM, FP, and DM. For simplicity, we neglect the mapping of communication channels and assume that they are mapped onto the same core as the sender process, even though the proposed framework can deal with cases that are more general. The extension to mapping communication channels is obvious and only augments the dimensionality of the design space.

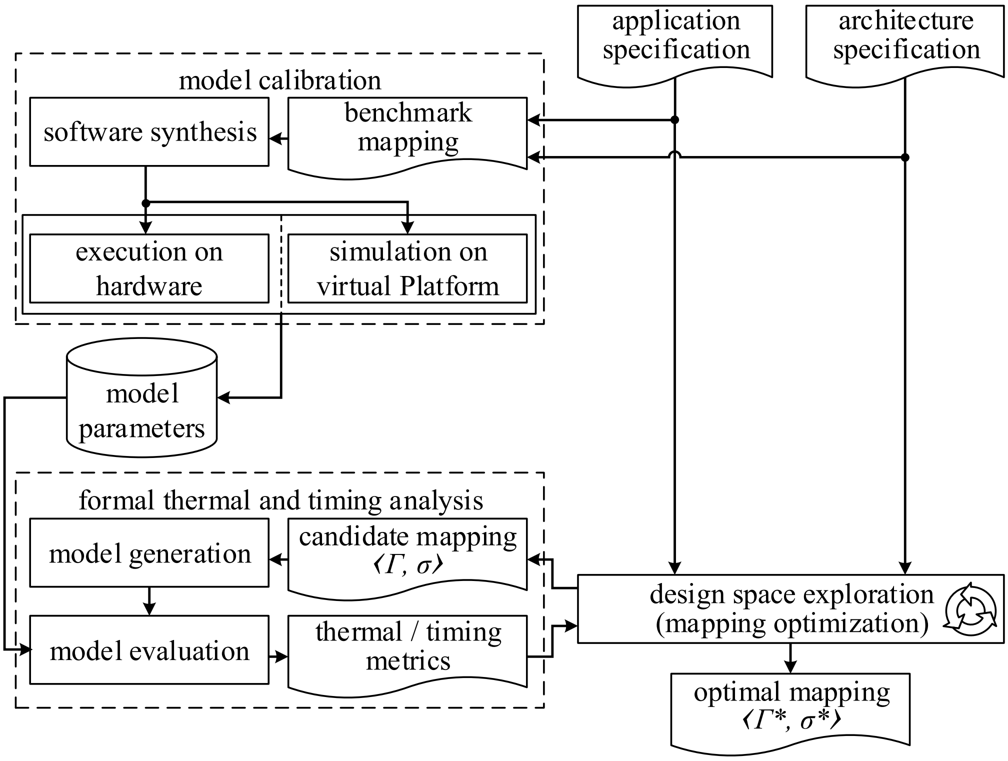


Fig. 5.3: Overview of the proposed thermal-aware mapping optimization framework.

5.3.4 Proposed Mapping Framework

The goal of the proposed framework is to explore the optimal mapping, i.e., the optimal binding and scheduling of a process network onto a many-core SoC platform with respect to both worst-case performance and worst-case chip temperature. By varying the binding of application elements, i.e., processes $v \in V$ and channels $q \in Q$, to computation and communication resources, selected system properties are optimized. Figure 5.3 illustrates the necessary steps to calculate the optimized mapping automatically.

The framework is composed of three major parts, namely model calibration, design space exploration, and thermal and timing analysis. During design space exploration, the thermal and timing characteristics of every candidate mapping $\langle \Gamma, \sigma \rangle$ are analyzed. The corresponding analysis models are parameterized with precalculated model parameters that are extracted during model calibration. The output of the considered framework is the optimal mapping $\langle \Gamma^*, \sigma^* \rangle$ of a given process network p onto a many-core SoC architecture $\mathcal{A}$.

The acquisition of the required model parameters is denoted as model calibration and is detailed in Section 5.6. Model calibration is performed based on a set of benchmark mappings prior to the analysis of the candidate mappings. First, a benchmark implementation is generated by synthesizing the benchmark system composed of the application, architecture, and benchmark mapping specifications. Then, the benchmark implementation is simulated on a virtual platform or executed on the real hardware. The required parameters are extracted and stored in a database for later use during thermal and timing analysis.

Once the model parameters are extracted, the design space exploration tool can start to explore for optimal mappings by analyzing the performance and chip temperature of various candidate mappings $\langle \Gamma, \sigma \rangle$. The proposed design flow uses the MPA framework [WTVL06] for performance analysis based on formal worst-case real-time analysis methods. In order to evaluate the performance of a single candidate mapping, the proposed mapping optimization framework performs the following steps: First, the framework generates an abstract model out of the same set of specifications as it is used for software synthesis, namely application, architecture, and candidate mapping specifications. Then, in a second step, the abstract model is examined with respect to timing and thermal characteristics. Timing properties are analyzed with the methods described in [HHB$^+$12] and thermal properties with the methods proposed in Section 5.5. We argue in this chapter that the considered mapping optimization framework can be completely automated. In other words, after specifying the input, i.e., the application and the architecture, the framework calculates a good mapping $\langle \Gamma^*, \sigma^* \rangle$ without user interaction.

5.4 System Model

This section introduces the formal models to analyze a streaming application on a many-core SoC platform with respect to both timing and temperature.

Notation: Bold characters are used for vectors and matrices and non-bold characters for scalars. For example, $\mathbf{H}$ denotes a matrix whose (k, ℓ)-th element is denoted $H_{k\ell}$ and $\mathbf{T}$ denotes a vector whose k-th element is denoted T_k.

5.4.1 Computational Model

The computational model for timing and thermal analysis is based on MPA [WTVL06] that uses a compositional approach to split a system up into actors with small interference. After characterizing each actor separately, the system is analyzed with real-time calculus [TCN00], which

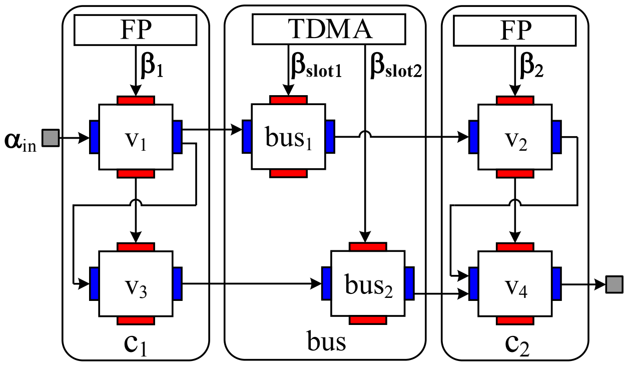


Fig. 5.4: MPA model of the exemplified system shown in Fig. 5.2. α_{in} abstracts the system's input stream, β_1 and β_2 abstract the available resources of core c_1 and core c_2. β_{slot_1} and β_{slot_2} abstract the resource availability of the bus.

is in turn based on network calculus [LBT01]. We define an MPA model as follows.

Def. 5.3: **(MPA Model)** *An MPA model is a graph $\mathcal{M} = \langle \mathcal{V}, E \rangle$. The set of nodes $\mathcal{V}$ represents the actors and the set of edges E represents event streams, abstracted by arrival curves α, and resource streams, abstracted by service curves β.*

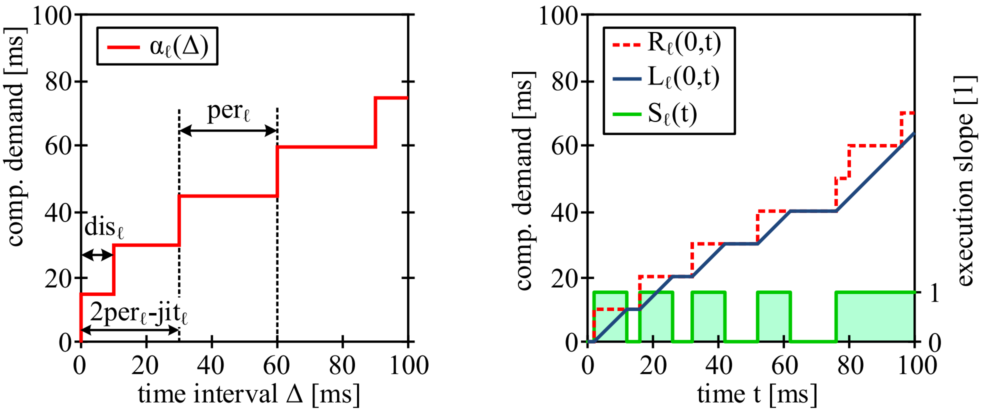
The data dependency between actors is given by the execution sequence. Communication resources are described by the same concept as computational resources, namely by a cumulative function that defines the number of available resources in any time interval.

Ex. 5.2: *Figure 5.4 shows the MPA model of the exemplified system shown in Fig. 5.2. For illustration, we suppose fixed priority preemptive scheduling on all cores and Time Division Multiple Access (TDMA) scheduling on the bus even though other policies can be modeled, as well. α_{in} abstracts the system's input stream, β_1 and β_2 abstract the available resources of core c_1 and core c_2. β_{slot_1} and β_{slot_2} abstract the resource availability of the bus.*

In the following, we will summarize the event stream and workload models relevant to calculate the worst-case chip temperature of a many-core SoC.

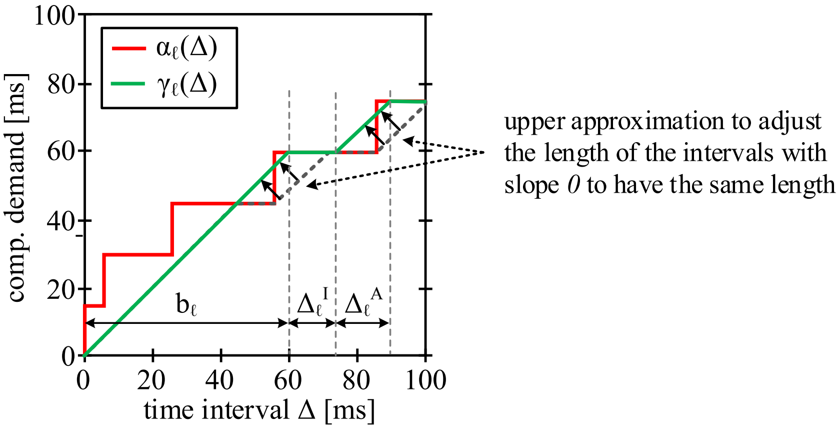
The considered model of an event stream follows the ideas of real-time calculus. We suppose that in time interval $[s, t)$, events with a total workload of $R_\ell(s, t)$ arrive at core c_ℓ. Each event is supposed to have a constant workload of Δ_ℓ^A. The arrival curve α_ℓ upper bounds all possible cumulative workloads:

$$R_\ell(s, t) \le \alpha_\ell(t - s) \quad \forall s < t \tag{5.2}$$



(a) Arrival curve α_ℓ that corresponds to an event stream defined by the period per_ℓ, the jitter jit_ℓ, and the minimum interarrival distance dis_ℓ.

(b) Typical cumulated workload curve $R_\ell(0,t)$ with the resulting accumulated computing time $L_\ell(0,t)$ and the associated mode function $S_\ell(t)$.



(c) Arrival curve α_ℓ and the upper bound curve $\gamma_\ell(\Delta)$ on the corresponding accumulated computing time. b_ℓ is the length of the first interval with slope 1, also called *burst*, Δ_ℓ^A is the length of every other interval with slope 1, and Δ_ℓ^I is the length of every interval with slope 0.

Fig. 5.5: Illustrations of the considered computational model.

with $\alpha_\ell(0) = 0$. Fig. 5.5a illustrates the concept of arrival curves for the widely-used standard event model where the event stream is defined by the parametric triple $\langle per_\ell, jit_\ell, dis_\ell \rangle$ with period per_ℓ, jitter jit_ℓ, and minimum interarrival distance dis_ℓ [HHJ+05]. For the remainder of this chapter, we assume that an event stream is always characterized by these three parameters.

We suppose that the processing cores are work-conserving. In other words, they will be in 'active' mode as long as there are events in their ready queues. The resource availability of core c_ℓ is abstracted by service curve $\beta_\ell(\Delta) = \Delta$ for all intervals of length $\Delta \geq 0$. The accumulated computing time $L_\ell(s,t)$ describes the amount of time units that core c_ℓ is spending to process an incoming workload of $R_\ell(s,t)$. Therefore, for work-conserving scheduling algorithms, the accumulated computing time $L_\ell(s,t)$ in time interval $[s,t)$ is:

$$L_\ell(s,t) = \inf_{s \leq u \leq t} \{(t-u) + R_\ell(s,u)\} \tag{5.3}$$

provided that there is no buffered workload in the ready queue at time s [TCN00].

For any fixed s with $s < t$, the accumulated computing time $L_\ell(s,t)$ is monotonically increasing and has either slope 1 or 0. Whenever the slope is 1, the core is in 'active' processing mode, i.e., it is processing events. When the core is idle, i.e., in sleep mode, the slope is 0. Thus, we can express the processing mode by the mode function:

$$S_\ell(t) = \frac{\mathrm{d}L_\ell(s,t)}{\mathrm{d}t} = \begin{cases} 1 & \text{core } c_\ell \text{ is 'active',} \\ 0 & \text{core } c_\ell \text{ is 'idle'.} \end{cases} \tag{5.4}$$

In Fig. 5.5b, the computing model is illustrated by comparing a typical cumulated workload curve, the resulting accumulated computing time, and the associated mode function.

Using arrival curve α_ℓ, the accumulated computing time $L_\ell(t-\Delta,t)$ can be upper bounded by $\gamma_\ell(\Delta)$ for all intervals of length $\Delta < t$:

$$L_\ell(t-\Delta,t) \leq \gamma_\ell(\Delta) = \inf_{0 \leq \lambda \leq \Delta} \{(\Delta-\lambda) + \alpha_\ell(\lambda)\}\,. \tag{5.5}$$

We characterize the upper bound on the accumulated computing time of core c_ℓ by the length b_ℓ of the first interval with slope 1, also called *burst*, the length Δ_ℓ^A of every other interval with slope 1, and the length Δ_ℓ^I of every interval with slope 0. While b_ℓ and Δ_ℓ^A are constant for the considered computational model, we also assume for computational simplicity that the upper bound on the accumulated computing time is selected so that all intervals with slope 0 have the same length. When calculating the critical computing time in Section 5.5, we use this simplicity to vary only the position of the burst and the gap between burst and the first active interval (see Algorithm 5.1). Otherwise, the positions of the additional gaps would have to be varied, as well. As every tight upper bound can be transformed in such an upper bound by slightly increasing the jitter, and the length of the burst is usually much larger than the length of a non-increasing interval, this assumption leads only to a small overapproximation of the worst-case chip temperature. Figure 5.5c illustrates the calculation of the upper bound on the accumulated computing time function.

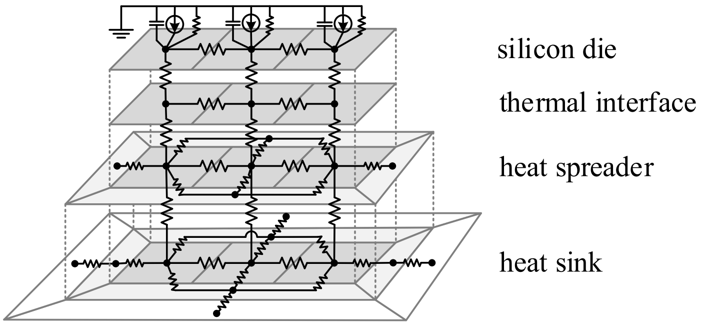


Fig. 5.6: Example *RC* circuit with three cores, and therefore, 24 nodes. For clarity, the structure is shown upside-down. Each node is connected by an additional resistance, a capacitance, and a current source with ground (in this figure, this is only shown for the silicon die layer).

5.4.2 Thermal Model

A well-accepted thermal model of a modern processor architecture is to describe the temperature evolution by means of an equivalent *RC* circuit [Kre99, SSS$^+$04, HGV$^+$06, CDH08]. For instance, the HotSpot model introduced in [SSS$^+$04, HGV$^+$06] uses four vertical layers, namely the heat sink, heat spreader, thermal interface, and silicon die, to model the thermal behavior of a processor. Each layer is divided further into a set of blocks according to architecture-level units and additional trapezoid blocks are introduced in the heat spreader and heat sink layer to model the area not covered by the subjacent layer. Every block is then mapped to a node of the thermal circuit so that the heat flow corresponds to the current passing through a thermal resistance, which generates a temperature difference analogous to a voltage. Finally, the power dissipation is modeled by connecting a current source to every node. For the scope of this thesis, we assume that every architectural unit is a processing core even though the method can be applied to a finer granularity without modifications. Figure 5.6 shows the *RC* circuit of a chip with three cores and therefore 24 nodes.

The η-dimensional temperature vector $\mathbf{T}(t)$ at time t is therefore described by a set of first-order differential equations:

$$\boldsymbol{C} \cdot \frac{\mathrm{d}\mathbf{T}(t)}{\mathrm{d}t} = \left(\boldsymbol{\mathcal{P}}(t) + \mathbf{K} \cdot \mathbf{T}^{\mathrm{amb}}\right) - (\mathbf{G} + \mathbf{K}) \cdot \mathbf{T}(t) \tag{5.6}$$

where η is the number of nodes of the *RC* circuit, $\boldsymbol{C}$ is the thermal capacitance matrix, $\mathbf{G}$ is the thermal conductance matrix, $\mathbf{K}$ is the thermal ground conductance matrix, $\boldsymbol{\mathcal{P}}$ is the power dissipation vector, and $\mathbf{T}^{\mathrm{amb}} = T^{\mathrm{amb}} \cdot [1, \ldots, 1]'$ is the ambient temperature vector. The initial temperature vector is denoted by $\mathbf{T}^0$ and the system is assumed to start at time $t^0 = 0$. $\boldsymbol{C}$, $\mathbf{G}$, and $\mathbf{K}$ are calculated from the floorplan and thermal configuration of the chip. $\mathbf{G}$ is a non-positive matrix whose diagonal elements are zero and $\mathbf{K}$ is a non-negative diagonal matrix [SSS+04].

We assume a linear dependency of power dissipation on temperature due to leakage power [LDSY07, CDH08]. Nodes that do not correspond to an architectural unit have zero power dissipation, i.e., $\mathcal{P}_\ell = 0$. All other nodes have two processing modes, namely 'active' if the associated core is processing events, and 'idle' if the associated core is in sleep mode. If core c_ℓ is in 'active' processing mode at time t, the mode function $S_\ell(t)$ defined in Eq. (5.4) is 1 and otherwise, $S_\ell(t) = 0$. As we suppose that the leakage power is independent of its processing mode, the power dissipation is given by:

$$\boldsymbol{\mathcal{P}}(t) = \boldsymbol{\phi} \cdot \mathbf{T}(t) + \boldsymbol{\psi}(t) \tag{5.7}$$

with:

$$\mathcal{P}_\ell(t) = \begin{cases} \mathcal{P}^a_\ell(t) = \phi_{\ell\ell} \cdot T_\ell(t) + \psi^a_\ell & \text{if } S_\ell(t) = 1, \\ \mathcal{P}^i_\ell(t) = \phi_{\ell\ell} \cdot T_\ell(t) + \psi^i_\ell & \text{if } S_\ell(t) = 0, \end{cases} \tag{5.8}$$

and $\boldsymbol{\phi}$ being a diagonal matrix with constant coefficients and $\boldsymbol{\psi}$ being a vector with constant coefficients.

Rewriting Eq. (5.6) with Eq. (5.7) leads to the state-space representation of the thermal model:

$$\frac{\mathrm{d}\mathbf{T}(t)}{\mathrm{d}t} = \mathbf{A} \cdot \mathbf{T}(t) + \mathbf{B} \cdot \mathbf{u}(t) \tag{5.9}$$

where $\mathbf{u}(t) = \boldsymbol{\psi}(t) + \mathbf{K} \cdot \mathbf{T}^{\mathrm{amb}}$ is called the input vector, $\mathbf{A} = -\boldsymbol{C}^{-1} \cdot (\mathbf{G} + \mathbf{K} - \boldsymbol{\phi})$, and $\mathbf{B} = \boldsymbol{C}^{-1}$. As $\mathbf{A}$ and $\mathbf{B}$ are time-invariant, the considered thermal model represents a Linear and Time-Invariant (LTI) system [Fri86] and consequently, for $t > 0$, a closed-form solution of the temperature yields:

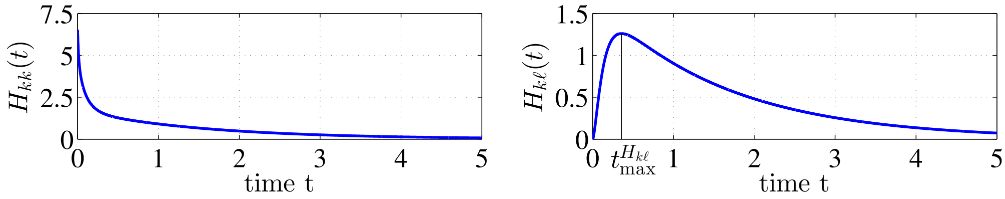
$$\mathbf{T}(t) = e^{\mathbf{A} \cdot t} \cdot \mathbf{T}^0 + \int_{-\infty}^{\infty} \mathbf{H}(t - \xi) \cdot \mathbf{u}(\xi) \, \mathrm{d}\xi \tag{5.10}$$

where $\mathbf{H}(t) = e^{\mathbf{A} \cdot t} \cdot \mathbf{B}$. $H_{k\ell}(t)$ corresponds to the impulse response between node ℓ and node k. With $\mathbf{T}^{\mathrm{init}}(t) = e^{\mathbf{A} \cdot t} \cdot \mathbf{T}^0$, the temperature $T_k(t)$ of node k is of form:

$$T_k(t) = T_k^{\mathrm{init}}(t) + \sum_{\ell=1}^{\eta} T_{k,\ell}(t) \tag{5.11}$$

where $T_{k,\ell}(t)$ is the convolution between impulse response $H_{k\ell}$ and input u_ℓ:

$$T_{k,\ell}(t) = \int_0^t H_{k\ell}(t - \xi) \cdot u_\ell(\xi) \, \mathrm{d}\xi. \tag{5.12}$$



(a) Self-impulse response $H_{kk}(t)$. **(b)** General impulse response $H_{k\ell}(t)$.

Fig. 5.7: Examples of two impulse responses. In general, the impulse response describes the thermal reaction of the system over time.

By using the processing mode of processing core c_ℓ, which is associated with node ℓ, we can connect input u_ℓ of node ℓ with the workload of the associated core:

$$u_\ell(t) = S_\ell(t) \cdot u_\ell^a + (1 - S_\ell(t)) \cdot u_\ell^i \tag{5.13}$$

where $u_\ell^a = \psi_\ell^a + K_{\ell\ell} \cdot T^{\text{amb}}$ and $u_\ell^i = \psi_\ell^i + K_{\ell\ell} \cdot T^{\text{amb}}$.

As the impulse response describes the reaction of the system over time, next, we will discuss its properties with respect to the considered thermal model. First, we note that $\mathbf{H}(t) \geq \mathbf{0}$ for all $t \geq 0$, as $\mathbf{A}$ is essentially non-negative, i.e., $\mathbf{A}_{k\ell} \geq 0$ for all $\ell \neq k$, which in turn leads to $e^{\mathbf{A}\cdot t} \geq \mathbf{0}$ [BV58]. The self-impulse response $H_{kk}(t)$ can be calculated by $H_{kk}(t) = e_k' \cdot e^{\mathbf{A}\cdot t} \cdot e_k \cdot B_{kk}$, where e_k is the unit vector pointing in k direction. Therefore, $H_{kk}(0) \geq H_{kk}(t) \geq 0$ for all $t \geq 0$ and $\frac{\mathrm{d}H_{kk}(t)}{\mathrm{d}t} \leq 0$ [MKK77], see Fig. 5.7a for an illustration.

Next, based on various experiments and the self-impulse response, we conjecture that the general impulse response $H_{k\ell}(t)$ is a non-negative unimodal[2] function that has its maximum at time $t_{\text{max}}^{H_{k\ell}}$, i.e., $\frac{\mathrm{d}H_{k\ell}(t)}{\mathrm{d}t} \geq 0$ for all $0 \leq t \leq t_{\text{max}}^{H_{k\ell}}$ and $\frac{\mathrm{d}H_{k\ell}(t)}{\mathrm{d}t} \leq 0$ for all $t > t_{\text{max}}^{H_{k\ell}}$ as illustrated in Fig. 5.7b. Intuitively, this can be motivated by the duality of a thermal network and a grounded capacitor RC circuit. We know from [GTP97] that all impulse responses of a stable RC tree network are unimodal. As a particular path of a general RC network usually dominates the impulse response, we can neglect local maximums that are caused by different paths without affecting the resulting temperature significantly. In all performed experiments, we never detected a departure from this conjecture. In summary, it can be said that temperature rises with power on the core that produces power without a delay and on a neighbor of the core that produces power only after a delay.

---

[2] A function $g(t)$ is called unimodal if and only if there exists some value $t = t^*$ such that $g(t)$ is nondecreasing for $t < t^*$ and nonincreasing for $t > t^*$.

5.5 Peak Temperature Analysis

The framework presented in this chapter uses formal thermal analysis methods to calculate the worst-case chip temperature, i.e., the maximum temperature of a chip under all feasible scenarios of task arrivals.

In the following, the corresponding thermal analysis methods are introduced. We start by presenting a constructive method to obtain the critical computing time and the critical workload leading to the worst-case chip temperature. As this method might be time-consuming and hence not suited for design space exploration, we will derive an analytical expression for an upper bound on the worst-case chip temperature in the second half of this section.

5.5.1 Preliminaries

The worst-case chip temperature T_S^* of a many-core platform is the maximum possible temperature of all nodes:

$$T_S^* = \max\left(T_1^*, \ldots, T_\eta^*\right) \tag{5.14}$$

where T_k^* is the worst-case peak temperature of node k and η the number of nodes. Because of non-determinism in the workload arriving at the different processing cores, one has to identify the *critical* set of cumulative workload traces $\mathbf{R}$ that leads to the worst-case peak temperature T_k^* of node k. We denote this critical set of cumulative workload traces by $\mathbf{R}^{\{k\}}$. Note that T_k^* does not only depend on the workload of core k, but also on the workload of all other cores of the chip due to the heat exchange between neighboring cores. In fact, because temperature rises with power consumed at another core only after a delay and the delay is different for every two cores, there is a different critical set of workload traces $\mathbf{R}^{\{k\}}$ for every node k.

In the first part of this section, we present a constructive method to calculate $\mathbf{R}^{\{k\}}$. We start by calculating the critical accumulated computing time $\mathbf{L}^{\{k\}}$ leading to T_k^*. Then, we show that $R_\ell^{\{k\}}(0,t) = \Delta_\ell^A \cdot \left\lceil L_\ell^{\{k\}}(0,t)/\Delta_\ell^A \right\rceil$ is a valid workload and $\mathbf{R}^{\{k\}}(0,t) = \left[R_1^{\{k\}}(0,t), \ldots, R_\eta^{\{k\}}(0,t)\right]'$ actually leads to the critical accumulated computing time $\mathbf{L}^{\{k\}}$. Finally, the worst-case peak temperature $T_k^*(\tau)$ of node k at observation time τ is obtained by simulating the system with workload $\mathbf{R}^{\{k\}}(0,t)$ for all $t \in [0,\tau)$.

We start by constructing the accumulated computing time $\mathbf{L}^{\{k\}}$ that maximizes temperature $T_k(\tau)$ at a certain observation time $\tau > 0$. In a first step, we show that each $T_{k,\ell}^*(\tau)$ defined as in Eq. (5.12) can be maximized individually.

Lem. 5.1: **(Superposition)** *Suppose that* $T^*_{k,\ell}(\tau) = \max_{u_\ell \in U_\ell}(T_{k,\ell}(\tau))$ *with* U_ℓ *the set of all possible inputs* u_ℓ*. Then, the worst-case peak temperature of node k at time* τ *is:*

$$T^*_k(\tau) \leq T^{\text{init}}_k + \sum_{\ell=1}^{\eta} T^*_{k,\ell}(\tau) \tag{5.15}$$

where η *is the number of nodes of the thermal RC circuit. Equality is obtained if the workloads of all processing cores are independent of each other.*

Proof. From Eq. (5.11) we obtain:

$$\begin{aligned} T^*_k(\tau) &= \max_{\mathbf{u}\in\mathbf{U}}(T_k(\tau)) = \max_{\mathbf{u}\in\mathbf{U}}\left(T^{\text{init}}_k + \sum_{\ell=1}^{\eta} T_{k,\ell}(\tau)\right) \\ &\leq T^{\text{init}}_k + \sum_{\ell=1}^{\eta}\max_{u_\ell\in U_\ell}(T_{k,\ell}(\tau)) = T^{\text{init}}_k + \sum_{\ell=1}^{\eta} T^*_{k,\ell}(\tau). \end{aligned} \tag{5.16}$$

With equality if $u_1, \ldots u_\eta$ are mutually independent, i.e., the workload of all cores is independent of each other. □

Lemma 5.1 indicates that each $T_{k,\ell}(\tau)$, at a certain time instance $\tau > 0$, can be maximized individually. As $T^*_{k,\ell}$ only depends on $L^{\{k\}}_\ell$ and $L^{\{k\}}_\ell$ only affects $T^*_{k,\ell}$, we can calculate every $L^{\{k\}}_\ell$ individually so that $L^{\{k\}}_\ell$ maximizes $T_{k,\ell}(\tau)$ at time instance τ.

5.5.2 Critical Accumulated Computing Time

Section 5.5.1 shows that each critical accumulated computing time $L^{\{k\}}_\ell$ can be calculated individually. In the following, we introduce a method to do so, i.e., we present a method to construct the critical accumulated computing time $L^{\{k\}}_\ell$ for each individual core c_ℓ.

Suppose that $t^{H_{k\ell}}_{\max}$ denotes the time where $H_{k\ell}(t)$ is maximum. First, we will show that a higher temperature is obtained at time instance τ if the core is in 'active' processing mode for a longer accumulated computing time in any time interval starting or ending at $\tilde{t}^{H_{k\ell}}_{\max} = \tau - t^{H_{k\ell}}_{\max}$. We will prove this statement first for $t < \tilde{t}^{H_{k\ell}}_{\max}$ (Lemmata 5.2a and 5.3a) and afterwards for $t > \tilde{t}^{H_{k\ell}}_{\max}$ (Lemmata 5.2b and 5.3b). Afterwards, we will refine these properties in Lemma 5.4 for the considered model of computation. Finally, Theorem 5.1 constructs the critical accumulated computing time $L^{\{k\}}_\ell$ for the considered workload model.

More detailed, Lemmata 5.2a and 5.2b support Lemmata 5.3a and 5.3b by showing that two mode functions that are identical except in a small interval result in different temperatures. Lemmata 5.3a and 5.3b use the results of Lemmata 5.2a and 5.2b and define the conditions for a higher temperature if the mode function is different at arbitrary positions. The mode function $S_\ell(t) = \frac{\mathrm{d}L_\ell(s,t)}{\mathrm{d}t}$ of core c_ℓ is defined in as Eq. (5.4) and is the derivation of the accumulated computing time function $L_\ell(s,t)$. In other words, $S_\ell(t) = 1$ if the core is in 'active' processing mode at time t and $S_\ell(t) = 0$ if the core is in 'idle' processing mode at time t.

Lem. 5.2a: **(Shifting, $t < \tilde{t}_{\max}^{H_{k\ell}}$)** *For any given time instance τ, we consider two mode functions $S_\ell(t)$ and $\overline{S}_\ell(t)$ defined as in Eq.* (5.4) *for $t \in [0,\tau)$. For given $\delta > 0, \sigma \geq 0, \sigma + 2\delta < \tilde{t}_{\max}^{H_{k\ell}}$, the two mode functions only differ as follows:*

- *$S_\ell(t) = 1$ for all $t \in [\sigma, \sigma+\delta)$ ('active mode'),*
- *$S_\ell(t) = 0$ for all $t \in [\sigma+\delta, \sigma+2\delta)$ ('idle mode'), and*
- *$\overline{S}_\ell(t) = 1 - S_\ell(t)$ for all $t \in [\sigma, \sigma+2\delta)$.*

In other words, both mode functions have the same sequence of 'active' and 'idle' modes for $t \in [0,\sigma)$ and $t \in [\sigma+2\delta, \tau)$. Then, $\overline{T}_{k,\ell}(\tau)$ at time τ for mode function $\overline{S}_\ell(t)$ is not less than $T_{k,\ell}(\tau)$ at time τ for mode function $S_\ell(t)$, i.e., $\overline{T}_{k,\ell}(\tau) \geq T_{k,\ell}(\tau)$.

Proof. Rewriting Eq. (5.12) with Eq. (5.13) leads to:

$$T_{k,\ell}(t) = \Psi + (u_\ell^a - u_\ell^i) \cdot \int_0^t S_\ell(\xi) \cdot H_{k\ell}(t-\xi)\,\mathrm{d}\xi \tag{5.17}$$

and

$$\overline{T}_{k,\ell}(t) = \Psi + (u_\ell^a - u_\ell^i) \cdot \int_0^t \overline{S}_\ell(\xi) \cdot H_{k\ell}(t-\xi)\,\mathrm{d}\xi \tag{5.18}$$

with $\Psi = u_\ell^i \cdot \int_0^t H_{k\ell}(t-\xi)\,\mathrm{d}\xi$. As the mode function $S_\ell(t)$ only differs from $\overline{S}_\ell(t)$ in time interval $[\sigma, \sigma+2\delta)$, we find:

$$\begin{aligned}&\left(\overline{T}_{k,\ell}(\tau) - T_{k,\ell}(\tau)\right)/(u_\ell^a - u_\ell^i)\\ &= \int_\sigma^{\sigma+2\delta} \overline{S}_\ell(t) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi - \int_\sigma^{\sigma+2\delta} S_\ell(t) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi.\end{aligned} \tag{5.19}$$

As $S_\ell(t) = 1$ for $t \in [\sigma, \sigma+\delta)$ and $S_\ell(t) = 0$ for $t \in [\sigma+\delta, \sigma+2\delta)$, and $\overline{S}_\ell(t) = 0$ for $t \in [\sigma, \sigma+\delta)$ and $\overline{S}_\ell(t) = 1$ for $t \in [\sigma+\delta, \sigma+2\delta)$, we find:

$$\begin{aligned}&\left(\overline{T}_{k,\ell}(\tau) - T_{k,\ell}(\tau)\right)/(u_\ell^a - u_\ell^i)\\ &= \int_{\sigma+\delta}^{\sigma+2\delta} H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi - \int_\sigma^{\sigma+\delta} H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi.\end{aligned} \tag{5.20}$$

As $H_{k\ell}(\tau - t)$ monotonically increases from 0 to $\tilde{t}_{\max}^{H_{k\ell}}$ and $u_\ell^a \geq u_\ell^i$, we get $\overline{T}_{k,\ell}(\tau) - T_{k,\ell}(\tau) \geq 0$. □

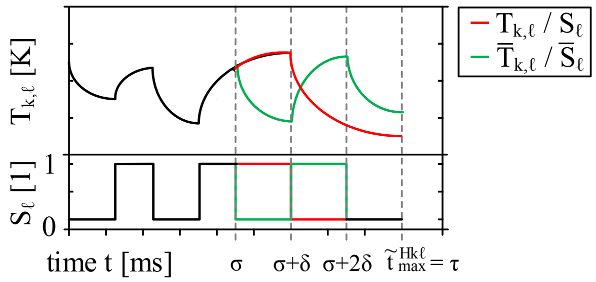


Fig. 5.8: Illustration of Lemma 5.2a for $\tilde{t}^{H_{k\ell}}_{\max} = \tau$. As the mode function $\overline{S}_\ell(t)$ is 'active' later, it leads to a higher temperature.

Lem. 5.2b: **(Shifting, $t > \tilde{t}^{H_{k\ell}}_{\max}$)** *For any given time instance τ, we consider two mode functions $S_\ell(t)$ and $\overline{S}_\ell(t)$ defined as in Eq.* (5.4) *for $t \in [0, \tau)$. For given $\delta > 0, \sigma \geq \tilde{t}^{H_{k\ell}}_{\max}, \sigma + 2\delta < \tau$, the two mode functions only differ as follows:*

- $\overline{S}_\ell(t) = 1$ *for all $t \in [\sigma, \sigma + \delta)$ ('active mode'),*
- $\overline{S}_\ell(t) = 0$ *for all $t \in [\sigma + \delta, \sigma + 2\delta)$ ('idle mode'), and*
- $S_\ell(t) = 1 - \overline{S}_\ell(t)$ *for all $t \in [\sigma, \sigma + 2\delta)$.*

In other words, both mode functions have the same sequence of 'active' and 'idle' modes for $t \in [0, \sigma)$ and $t \in [\sigma + 2\delta, \tau)$. Then, $\overline{T}_{k,\ell}(\tau)$ at time τ for mode function $\overline{S}_\ell(t)$ is not less than $T_{k,\ell}(\tau)$ at time τ for mode function $S_\ell(t)$, i.e., $\overline{T}_{k,\ell}(\tau) \geq T_{k,\ell}(\tau)$.

Proof. We omit the proof as it is similar to the one of Lemma 5.2a □

Figure 5.8 illustrates Lemma 5.2a for $\tilde{t}^{H_{k\ell}}_{\max} = \tau$. Now, we will show that a higher temperature at time τ is obtained if in any time interval starting or ending at $\tilde{t}^{H_{k\ell}}_{\max}$, the core is in 'active' processing mode for a longer accumulated time. Lemmata 5.3a and 5.3b will discuss this statement for intervals ending and starting at time $\tilde{t}^{H_{k\ell}}_{\max}$, respectively.

Lem. 5.3a: **(Mode Functions Comparison, $t < \tilde{t}^{H_{k\ell}}_{\max}$)** *For any given time instance τ, we consider two accumulated computing time functions L_ℓ, resulting from mode function S_ℓ, and $\overline{L}_\ell$, resulting from mode function $\overline{S}_\ell$, with:*

$$\overline{L}_\ell(\tilde{t}^{H_{k\ell}}_{\max} - \Delta, \tilde{t}^{H_{k\ell}}_{\max}) \geq L_\ell(\tilde{t}^{H_{k\ell}}_{\max} - \Delta, \tilde{t}^{H_{k\ell}}_{\max}) \tag{5.21}$$

for all $0 \leq \Delta \leq \tilde{t}^{H_{k\ell}}_{\max}$ and $S_\ell(t) = \overline{S}_\ell(t)$ for all $\tilde{t}^{H_{k\ell}}_{\max} < t \leq \tau$. Then, $\overline{T}_{k,\ell}(\tau)$ at time τ for mode function $\overline{S}_\ell(t)$ is not less than $T_{k,\ell}(\tau)$ at time τ for mode function $S_\ell(t)$.

Proof. Equation (5.21) can be translated by means of Eq. (5.4) into $\int_{\tilde{t}_{\max}^{H_{k\ell}}-\Delta}^{\tilde{t}_{\max}^{H_{k\ell}}} \overline{S}_\ell(t)\,\mathrm{d}t \geq \int_{\tilde{t}_{\max}^{H_{k\ell}}-\Delta}^{\tilde{t}_{\max}^{H_{k\ell}}} S_\ell(t)\,\mathrm{d}t$. Then, the proof is equivalent to the proof of Lemma 3 of [RYB⁺11] and is therefore omitted. In particular, Lemma 2 of [RYB⁺11] is replaced with Lemma 5.2a and τ by $\tilde{t}_{\max}^{H_{k\ell}}$. □

Lem. 5.3b: (Mode Functions Comparison, $t > \tilde{t}_{\max}^{H_{k\ell}}$) *For any given time instance τ, we consider two accumulated computing time functions L_ℓ, resulting from mode function S_ℓ, and $\overline{L}_\ell$, resulting from mode function $\overline{S}_\ell$, with:*

$$\overline{L}_\ell(\tilde{t}_{\max}^{H_{k\ell}}, \Delta) \geq L_\ell(\tilde{t}_{\max}^{H_{k\ell}}, \Delta) \tag{5.22}$$

for all $\tilde{t}_{\max}^{H_{k\ell}} \leq \Delta \leq \tau$ and $S_\ell(t) = \overline{S}_\ell(t)$ for all $0 \leq t < \tilde{t}_{\max}^{H_{k\ell}}$. Then, $\overline{T}_{k,\ell}(\tau)$ at time τ for mode function $\overline{S}_\ell(t)$ is not less than $T_{k,\ell}(\tau)$ at time τ for mode function $S_\ell(t)$.

Proof. We omit the proof as it is similar to the one of Lemma 5.3a. □

So far, we have shown some properties of the critical accumulated computing time $L_\ell^{\{k\}}$. In particular, we have shown that a higher temperature at a time instance τ is obtained, if in any time interval starting or ending at $\tilde{t}_{\max}^{H_{k\ell}}$, the core is in 'active' processing mode for a longer accumulated time. Next, we will apply these properties to the considered model of computation, i.e., to event streams specified by a period, a jitter, and a minimum interarrival distance. We will show that a necessary condition to maximize the temperature at time τ is that the core is in 'active' processing mode during a time frame of at least $b_\ell - \Delta_\ell^A$ time units, where b_ℓ and Δ_ℓ^A are defined as in Section 5.4.1 and $\tilde{t}_{\max}^{H_{k\ell}}$ is within this time frame.

Lem. 5.4: (Burst) *Suppose that $L_\ell^{\{k\}}(0, \Delta)$ for all $0 \leq \Delta \leq \tau$ leads to an upper bound on $T^*_{k,\ell}(\tau)$ at time τ. When b_ℓ and Δ_ℓ^A are defined as in Section 5.4.1 and the scheduler is work-conserving, then there exist $t^{(l)}$ and $t^{(r)}$ such that $L_\ell^{\{k\}}(t^{(l)}, t^{(r)}) = t^{(r)} - t^{(l)} = b_\ell - \Delta_\ell^A$ and $\tilde{t}_{\max}^{H_{k\ell}} \in [t^{(l)}, t^{(r)}]$.*

Proof. For simplicity reasons, we only provide a visual and intuitive proof of this lemma. Suppose that $L_\ell(0, \Delta)$ for all $0 \leq \Delta \leq \tau$ resulting from S_ℓ leads to $T_{k,\ell}(\tau)$ and does not fulfill the condition stated in Lemma 5.4. We will transform $L_\ell(0, \Delta)$ into $\overline{L}_\ell(0, \Delta)$ resulting from $\overline{S}_\ell(t)$ that leads to $\overline{T}_{k,\ell}(\tau) > T_{k,\ell}(\tau)$.

First, we note that every $L_\ell(0, \Delta)$ can be transformed into $\overline{L}_\ell(0, \Delta)$ such that there exist t and Δ with $\gamma_\ell(\Delta) - \overline{L}_\ell(t - \Delta, t) = 0$, which we assume in the following. See, e.g., [Sch11] for a proof of this proposition.

Next, we compare two processing components with the same resource availability, namely $c_\ell^{(\gamma)}$ and $c_\ell^{(\omega)}$. The component $c_\ell^{(\gamma)}$ is processing an event

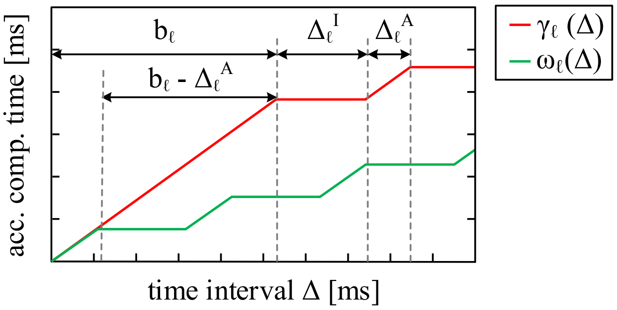


Fig. 5.9: Upper bound on the accumulated computing time of an event stream with jitter $\gamma_\ell(\Delta)$ and an event stream without jitter $\omega_\ell(\Delta)$.

stream leading to an accumulated computing time that is upper bounded by $\gamma_\ell(\Delta)$. $c_\ell^{(\omega)}$ is processing an event stream that has the same period as the event stream arriving at $c_\ell^{(\gamma)}$ but is free of jitter. We denote the upper bound on its accumulated computing time by $\omega_\ell(\Delta)$. Next, we will shift active processing modes closer to $\tilde{t}_{\max}^{H_{k\ell}}$ if L_ℓ exceeds $\omega_\ell(\Delta)$. In order to simplify the proof technicalities, we suppose discrete time, i.e., $S_\ell(t)$ may change values only at multiples of δ and is constant for $t \in [r \cdot \delta, (r+1) \cdot \delta)$ for all $r \geq 0$. Furthermore, $r^{(m)} = \tilde{t}_{\max}^{H_{k\ell}}/\delta$ and we execute the following algorithm:

1. Determine the smallest $r \in [1, r^{(m)}]$ such that $L_\ell(r \cdot \delta - \Delta, r \cdot \delta) > \omega_\ell(\Delta)$ for $\Delta > 0$. If there is no such r, then $S_\ell(t) = \overline{S}_\ell(t)$ and the algorithm stops.

2. Determine the smallest $r' \in (r, r^{(m)}]$ such that $S_\ell(r' \cdot \delta) = 0$. If such r' exists, then change $S_\ell(r \cdot \delta)$ from 1 to 0 and $S_\ell(r' \cdot \delta)$ from 0 to 1. Otherwise, the algorithm stops.

With the exception of the last iteration, $T_{k,\ell}(\tau)$ increases in every iteration as a result of Lemma 5.3a. Fig. 5.9 illustrates that c_ℓ^{γ} can process $b_\ell - \Delta_\ell^A$ time units more in 'active' mode as c_ℓ^{ω}. As the algorithm only switches the positions of these time units, Eq. (5.5) will never be violated. A similar algorithm can be performed for $t > \tilde{t}_{\max}^{H_{k\ell}}$, and therefore, there exist a $t^{(l)} \leq \tilde{t}_{\max}^{H_{k\ell}}$ and a $t^{(r)} \geq \tilde{t}_{\max}^{H_{k\ell}}$ such that $\overline{L}_\ell(t^{(l)}, t^{(r)}) = t^{(r)} - t^{(l)} = b_\ell - \Delta_\ell^A$ and $\overline{T}_{k,\ell}(\tau) \geq T_{k,\ell}(\tau)$. □

Based on the previous lemmata, we will show the first main result of this section. Theorem 5.1 provides a constructive algorithm to calculate the critical accumulated computing time $L_\ell^{\{k\}}(0, \Delta)$ for all $0 \leq \Delta \leq \tau$ that maximizes $T_{k,\ell}(\tau)$ at a certain observation time τ, i.e., $T_{k,\ell}^*(\tau) \geq T_{k,\ell}(\tau)$. In particular, Algorithm 5.1 calculates the critical accumulated computing time $L_\ell^{\{k\}}$ by varying both the position of the burst and the gap between burst and the first active interval. As sketched in Fig. 5.10, the critical accumulated computing time is the computing time that maximizes the sum of the areas below the impulse response curve where the core is in 'active' processing mode. b_ℓ, Δ_ℓ^I, and Δ_ℓ^A are defined as in Section 5.4.1 and the auxiliary function $\nu_\ell(t,\zeta)$ used in Algorithm 5.1 is one for Δ_ℓ^A time units, starting at time ζ.

Thm. 5.1: **(Critical Accumulated Computing Time)** *Suppose that function $\nu_\ell(t,\zeta)$ is defined as:*

$$\nu_\ell(t,\zeta) = \begin{cases} 1 & 0 \le \zeta \le t \le \min(\zeta + \Delta_\ell^A, \tau) \\ 0 & \textit{otherwise} \end{cases} \tag{5.23}$$

and the accumulated computing time function $L_\ell^{\{k\}}(0,\Delta)$ for all $0 \le \Delta \le \tau$ constructed by Algorithm 5.1 leads to $T_{k,\ell}^*(\tau)$ at time τ. When the scheduler is work-conserving, $T_{k,\ell}^*(\tau)$ is an upper bound on the highest value of $T_{k,\ell}(\tau)$ at time τ, i.e., $T_{k,\ell}^*(\tau) \ge T_{k,\ell}(\tau)$.

Algorithm 5.1 Calculation of the critical accumulated computing time function $L_\ell^{\{k\}}(0,\Delta)$ for all $0 \le \Delta \le \tau$.

```
Input: $b_\ell, \Delta_\ell^I, \Delta_\ell^A, per_\ell, \tilde{t}_{\max}^{H_{k\ell}}, \tau, H_{k\ell}$
Output: $L_\ell^{\{k\}}(0,\Delta)$
for all $t^{(r)}$ in $[\tilde{t}_{\max}^{H_{k\ell}}, \tilde{t}_{\max}^{H_{k\ell}} + b_\ell - \Delta_\ell^A]$ do    ▹find the position of the burst
    for all $t^s \in [0, \Delta_\ell^I]$ do    ▹find gap between burst and suc. active interval
        $t^{(l)} = t^{(r)} - b_\ell + \Delta_\ell^A$
        $S_\ell(t) = \begin{cases} 1 & t \in [t^{(r)} - b_\ell + \Delta_\ell^A, t^{(r)}) \\ 0 & \text{otherwise} \end{cases}$
        for $i = 1$ to $\left\lceil \frac{\tau - t^{(r)}}{per_\ell} \right\rceil$ do    ▹make trace for $t > t^{(r)}$
            $S_\ell(t) = S_\ell(t) + \nu_\ell\left(t, t^s + t^{(r)} + (i-1) \cdot per_\ell\right)$
        end for
        for $i = 1$ to $\left\lceil \frac{t^{(l)}}{per_\ell} \right\rceil$ do    ▹make trace for $t < t^{(l)}$
            $S_\ell(t) = S_\ell(t) + \nu_\ell\left(t, t^s + t^{(l)} - i \cdot per_\ell\right)$
        end for
        $T_{k,\ell} = \int_0^t S_\ell(\varphi) \cdot H_{k\ell}(t - \varphi)\, d\varphi$
        if $T_{k,\ell} > T_{k,\ell}^*$ then    ▹$T_{k,\ell}$ comparison
            $T_{k,\ell}^* = T_{k,\ell}$ and $S_\ell^{\{k\}} = S_\ell$
        end if
    end for
end for
$L_\ell^{\{k\}}(0,\Delta) = \int_0^\Delta S_\ell^*(\varphi)\, d\varphi$
```

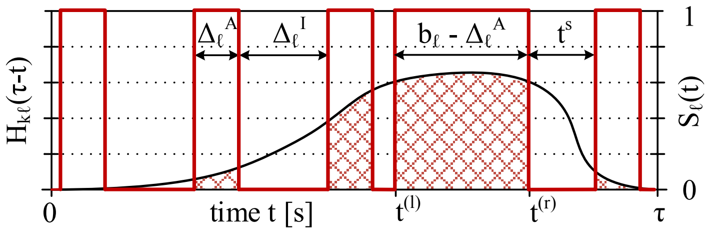


Fig. 5.10: Illustration of Algorithm 5.1 to calculate the critical accumulated computing time function $L_\ell^{\{k\}}(0, \Delta)$.

Proof. First, we show that $L_\ell^{\{k\}}(0, \Delta)$ resulting from $S_\ell^{\{k\}}$ and constructed by Algorithm 5.1 satisfies Eq. (5.5), i.e., it conforms to the upper bound on the accumulated computing time. With the exception of the burst, $S_\ell^{\{k\}}(t) = 1$ for Δ_ℓ^A time units in every interval of $per_\ell = \Delta_\ell^A + \Delta_\ell^I$ time units. As all blocks of 'active' processing mode have the same offset and the longest possible contiguous block of 'active' processing modes is b_ℓ time units, $L_\ell^{\{k\}}(t - \Delta, t) \leq \gamma_\ell(\Delta)$.

Now, let us prove that $T^*_{k,\ell}(\tau) \geq T_{k,\ell}(\tau)$. By construction, the condition stated in Lemma 5.4, i.e., there exist $t^{(l)}$ and $t^{(r)}$ such that $L^*_\ell(t^{(l)}, t^{(r)}) = t^{(r)} - t^{(l)} = b_\ell - \Delta_\ell^A$ and $\tilde{t}^{H_{k\ell}}_{\max} \in [t^{(l)}, t^{(r)}]$, is fulfilled. The remaining work is to show that the temperature is maximized if for $t \notin [t^{(l)}, t^{(r)}]$, the processing mode is alternately 'active' and 'idle' for Δ_ℓ^A and Δ_ℓ^I time units, respectively. However, this is a direct consequence of Lemma 5.3a and Lemma 5.3b. The offset t^s is required to guarantee that Eq. (5.5) is never violated. Now, $L_\ell^{\{k\}}$ can be calculated by iterating over the offset t^s of the periodic invocation and the start positions $t^{(r)}$ of the burst, which in turn proves the theorem. □

So far, we only derived the accumulated computing time that maximizes the temperature at observation time τ. However, we did not discuss the amount of observation time τ. Next, we will show that increasing the observation time τ will not decrease the worst-case peak temperature T_k^* if $\mathbf{T}^0 \leq (\mathbf{T}^\infty)^i$, where $(\mathbf{T}^\infty)^i$ is the steady-state temperature vector if all cores are in 'idle' mode.

Lem. 5.5: **(Initial Temperature)** *Suppose that the accumulated computing time function* $\mathbf{L}^{\{k\}}(0, \Delta)$ *for all* $0 \leq \Delta \leq \tau$ *from Theorem 5.1 leads to the worst-case peak temperature* $T_k^*(\tau)$ *at time* τ*. Then,* $T_k^*(\tau) \geq T_k(t)$ *for all* $0 \leq t \leq \tau$ *and for any set of feasible workload traces with the same initial temperature vector* $\mathbf{T}^0 \leq (\mathbf{T}^\infty)^i$.

Proof. As $H_{k\ell}(t) \geq 0$ for all t, k, and ℓ, it follows from Eq. (5.11) that the temperature caused by any $\mathbf{L}$ is never smaller than the temperature of a system that only operates in 'idle' mode. As $\mathbf{T}(0) = \mathbf{T}^0 \leq (\mathbf{T}^\infty)^i$, temperatures resulting from any $\mathbf{L}$ satisfy $T_k(t) \geq T_k(0)$ for $0 \leq t \leq \tau$, which, in particular, also holds for T_k^*, i.e., $T_k^*(t) \geq T_k^*(0)$ for $0 \leq t \leq \tau$.

Assume for contradiction that $T_k(\sigma) > T_k^*(\tau)$ for $\sigma \leq \tau$. Then there exists a ℓ so that $T_{k,\ell}(\sigma) > T_{k,\ell}^*(\tau)$. According to Theorem 5.1, there exists a $\overline{S}_\ell^{\{k\}}$ that maximizes $T_{k,\ell}(\sigma)$. However, by the linearity of the system, $\overline{S}_\ell^{\{k\}}(t-(\tau-\sigma))$ results in $T_{k,\ell}(\tau) = T_{k,\ell}(\sigma)$ for $\mathbf{T}_0 = (\mathbf{T}^\infty)^i$ and $T_{k,\ell}(\tau) \geq T_{k,\ell}(\sigma)$ for $\mathbf{T}^0 < (\mathbf{T}^\infty)^i$, which is a contradiction. □

5.5.3 Critical Accumulated Workload

Theorem 5.1 provides an upper bound $T_{k,\ell}^*(\tau)$ on $T_{k,\ell}(\tau)$ at a certain observation time τ. However, there might be no workload trace that leads to the critical accumulated computing time $L_\ell^{\{k\}}(0,\Delta)$. In the following, we will show that this is not the case, and $R_\ell^{\{k\}}(0,\Delta) = \Delta_\ell^A \cdot \left\lceil L_\ell^{\{k\}}(0,\Delta)/\Delta_\ell^A \right\rceil$ actually leads to the critical accumulated computing time.

Thm. 5.2: **(Critical Cumulated Workload)** *Suppose that the accumulated computing time function* $L_\ell^{\{k\}}(0,\Delta)$ *for all* $0 \leq \Delta \leq \tau$ *is defined as in Theorem 5.1. Then, the critical workload function* $R_\ell^{\{k\}}(0,\Delta) = \Delta_\ell^A \cdot \left\lceil L_\ell^{\{k\}}(0,\Delta)/\Delta_\ell^A \right\rceil$

- *leads to the accumulated computing time* $L_\ell^{\{k\}}(0,\Delta)$*, and*
- *complies with arrival curve* α_ℓ *according to Eq.* (5.2).

Furthermore, the set of workload functions $\mathbf{R}^{\{k\}}(0,t)$

- *leads to the highest possible temperature* $T_k^*(\tau) \geq T_k(t)$ *for all* $0 \leq t \leq \tau$ *and any set of feasible workload traces with the same initial temperature vector* $\mathbf{T}^0 \leq (\mathbf{T}^\infty)^i$.

Proof. Without loss of generality, we suppose that $\Delta_\ell^A = 1$. At first we show that the continuous workload function $\widehat{R}_\ell^{\{k\}}(0,\Delta) = L_\ell^{\{k\}}(0,\Delta)$ leads to $L_\ell^{\{k\}}(0,\Delta)$. As $\widehat{R}_\ell^{\{k\}}(0,\Delta) = L_\ell^{\{k\}}(0,\Delta)$, we have to prove that $L_\ell^{\{k\}}(0,\Delta) = \inf_{0\leq u\leq\Delta}\{(\Delta-u) + L_\ell^{\{k\}}(0,u)\}$. Because there exists a u' such that $(\Delta-u') + L_\ell^{\{k\}}(0,u') = (\Delta-u') + L_\ell^{\{k\}}(0,\Delta)$, namely $u' = \Delta$, we only have to show that $(\Delta-u) + L_\ell^{\{k\}}(0,u) \geq L_\ell^{\{k\}}(0,\Delta)$ for all $0 \leq u \leq \Delta$. This condition is equivalent to $(\Delta-u) \geq L_\ell^{\{k\}}(0,\Delta) - L_\ell^{\{k\}}(0,u) = L_\ell^{\{k\}}(u,\Delta)$ and from the fact that the accumulated computing time in interval $[u,\Delta)$ cannot exceed the available time $\Delta-u$ follows that $\widehat{R}_\ell^{\{k\}}(0,\Delta)$ leads to $L_\ell^{\{k\}}(0,\Delta)$. Now, we can show that $R_\ell^{\{k\}}(0,\Delta) = \lceil \widehat{R}_\ell^{\{k\}}(0,\Delta) \rceil = \lceil L_\ell^{\{k\}}(0,\Delta) \rceil$ leads to $L_\ell^{\{k\}}(0,\Delta)$. From the specification of the accumulated computing time follows that

$\gamma_\ell(\Delta)$ has either slope 1 or 0 and has an integer value if it has slope 0. As $\nu_\ell(t, t^s)$ define in Eq. (5.23) guarantees that $L_\ell^{\{k\}}(0,\Delta)$ has the same properties as $\gamma_\ell(\Delta)$, $R_\ell^{\{k\}}(0,\Delta)$ leads to the accumulated computing time function $\inf_{0\le u\le\Delta}\{(\Delta-u)+R_\ell^{\{k\}}(0,u)\} = \inf_{0\le u\le\Delta}\{(\Delta-u)+\lceil L_\ell^{\{k\}}(0,u)\rceil\} = \inf_{0\le u\le\Delta}\{(\Delta-u)+L_\ell^{\{k\}}(0,u)\} = L_\ell^{\{k\}}(0,\Delta)$.

For the second item, we have to show that $R_\ell^{\{k\}}(x,y) \le \alpha_\ell(y-x)$. First, we note that $\gamma_\ell(\Delta) = \inf_{0\le\lambda\le\Delta}\{(\Delta-\lambda)+\alpha_\ell(\lambda)\} \le \alpha_\ell(\Delta)$ for $\Delta > 0$. Now, we find $R_\ell^{\{k\}}(x,y) = R_\ell^{\{k\}}(0,y) - R_\ell^{\{k\}}(0,x) = \lceil L_\ell^{\{k\}}(0,y)\rceil - \lceil L_\ell^{\{k\}}(0,x)\rceil \le \lceil L_\ell^{\{k\}}(0,y) - L_\ell^{\{k\}}(0,x)\rceil = \lceil L_\ell^{\{k\}}(x,y)\rceil \le \lceil\gamma_\ell(y-x)\rceil \le \lceil\alpha_\ell(y-x)\rceil = \alpha_\ell(y-x)$ for all $x < y$.

The third item is a simple consequence of Lemmata 5.1 and 5.5. First, we see that $\mathbf{R}^{\{k\}}$ leads to the accumulated computing time function $\mathbf{L}^{\{k\}}$ and secondly, $\mathbf{L}^{\{k\}}$ leads to the highest temperature $T_k^*(\tau) \ge T_k(t)$ according to Lemma 5.5. □

Suppose that the workloads of all processing cores are independent of each other. Then, the upper bound $T_k^*(\tau)$ determined by Theorem 5.2 is tight as there exists a workload that leads to the critical accumulated computing time. Theorem 5.2 completed the derivation of the constructive method to calculate the worst-case peak temperature $T_k^*(\tau)$ of core k. Starting from the set of arrival curves $\boldsymbol{\alpha}(\Delta)$, we calculate $\boldsymbol{\gamma}(\Delta)$ using Eq. (5.5). Afterwards, Algorithm 5.1 constructs $L_\ell^{\{k\}}$ for every processing core c_ℓ, which in turn determines the critical sequence of 'active' and 'idle' modes. Finally, $T_k^*(\tau)$ can be found by solving the thermal model defined in Eq. (5.6) at time $t = \tau$.

The complexity of calculating the worst-case chip temperature T_S^* is $O(\eta^2 * \varsigma)$ with η the number of nodes of the thermal RC circuit. The factor ς reflects the time to execute Algorithm 5.1 and is inversely proportional to the selected time step.

5.5.4 Fast Temperature Evaluation

As indicated, Algorithm 5.1 might be time-consuming and hence not suited for design space exploration. Therefore, in the following, we will derive an analytical expression for the accumulated computing time that leads to a non-trivial upper bound on the peak temperature. Afterwards, we use the obtained computing time to propose an analytical expression for an upper bound on the worst-case peak temperature.

The next lemma simplifies Algorithm 5.1 so that it is constant in time and the resulting upper bound at observation time τ, $\widehat{T}_k^*(\tau)$, is not smaller than the worst-case peak temperature of node k. $\widehat{\mathbf{L}}^{\{k\}}$ denotes the critical accumulated computing time that leads to $\widehat{T}_k^*(\tau)$.

Lem. 5.6: **(Analytical Expression for the Computing Time)** *Suppose that the accumulated computing time function* $\widehat{\mathbf{L}}^{\{k\}}(0,\Delta) = \left[\widehat{L}_1^{\{k\}}(0,\Delta),\ldots,\widehat{L}_\eta^{\{k\}}(0,\Delta)\right]'$ *for all* $0 \leq \Delta \leq \tau$ *with:*

$$\widehat{L}_\ell^{\{k\}}(0,\Delta) = \begin{cases} \gamma(\tilde{t}_{\max}^{H_{k\ell}}) - \gamma(\tilde{t}_{\max}^{H_{k\ell}} - \Delta) & 0 \leq \Delta < \tilde{t}_{\max}^{H_{k\ell}} \\ \gamma(\tilde{t}_{\max}^{H_{k\ell}}) + \gamma(\Delta - \tilde{t}_{\max}^{H_{k\ell}}) & \tilde{t}_{\max}^{H_{k\ell}} \leq \Delta < \tau \end{cases} \tag{5.24}$$

leads to $\widehat{T}_k^*(\tau)$ *at time* τ. *When the scheduler is work-conserving,* $\widehat{T}_k^*(\tau)$ *is an upper bound on the highest value of temperature* $T_k(\tau)$ *at time* τ. *Furthermore, when* $(\mathbf{T}^\infty)^i$ *is the steady-state temperature vector if all nodes are in 'idle' mode,* $\widehat{T}_k^*(\tau) \geq T_k(t)$ *for all* $0 \leq t \leq \tau$ *and any set of feasible workload traces with the same initial temperature vector* $\mathbf{T}^0 \leq (\mathbf{T}^\infty)^i$.

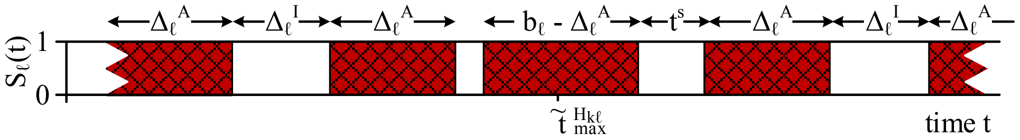
Proof. We will prove this lemma by translating $\mathbf{L}_\ell^{\{k\}}$ calculated with Algorithm 5.1 into $\widehat{\mathbf{L}}_\ell^{\{k\}}$ calculated with Eq. (5.24) and show that the temperature will not decrease in every step. To this end, we use the fact that the temperature does not decrease if the amount of 'active' time units per time interval is either increased or shifted closer to $\tilde{t}_{\max}^{H_{k\ell}}$ (Lemmata 5.2a and 5.2b).

In the first step, the precedent and successive active intervals of the burst are moved to the burst so that the node is continuously active for $b_\ell + \Delta_\ell^A$ time units, compare Figs. 5.11a and 5.11b for an illustration. The second step makes the search for the position of the burst obsolete. To this end, the length of the burst is extended so that it covers all possible positions of the burst, i.e., $\widehat{S}_\ell(t) = 1$ for all $t \in [\tilde{t}_{\max}^{H_{k\ell}} - b_\ell + \Delta_\ell^A, \tilde{t}_{\max}^{H_{k\ell}} + b_\ell - \Delta_\ell^A]$, see Fig. 5.11c for an illustration.

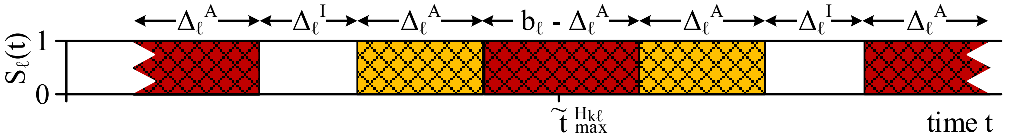
Algorithm 5.2 is the result of these two translations. One can readily prove that in both steps, the amount of 'active' time units is either increased or shifted closer to $\tilde{t}_{\max}^{H_{k\ell}}$. Finally, we note that Eq. (5.24) is equivalent to Algorithm 5.2, and therefore $\widehat{T}_k^*(\tau) \geq T_k^*(\tau)$. □

As the accumulated computing time function can be calculated in constant time, computing an upper bound on the worst-case chip temperature has time complexity $O(\eta^2)$ with η the number of nodes.

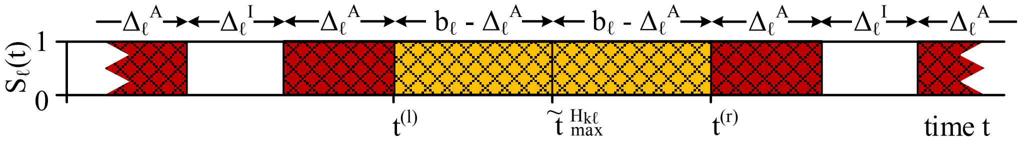
Next, we show that calculating the worst-case peak temperature can be simplified further by running processing core c_ℓ with constant slope $\delta_\ell = \frac{\Delta_\ell^A}{\Delta_\ell^A + \Delta_\ell^I}$ for all time units except during the burst. In fact, Δ_ℓ^A is the length of an interval with slope 1 and Δ_ℓ^I is the length of every interval with slope 0 of the upper bound on the accumulated computing time.



(a) Starting position corresponding to the critical accumulated computing time $L_\ell^{\{k\}}$.



(b) Translation step 1: the precedent and successive active intervals of the burst are moved to the burst.



(c) Translation step 2: the length of the burst is extended so that it covers all possible positions of the burst.

Fig. 5.11: Illustration of the steps to translate $L_\ell^{\{k\}}$ into $\widehat{L}_\ell^{\{k\}}$.

Algorithm 5.2 Calculation of the critical accumulated computing time function $\widehat{L}_\ell^{\{k\}}(0, \Delta)$ for all $0 \leq \Delta \leq \tau$ whereby function $\nu_\ell(t, \zeta)$ is defined as in Eq. (5.23).

```
Input: $b_\ell, \Delta_\ell^I, \Delta_\ell^A, per_\ell, \tilde{t}_{\max}^{H_{k\ell}}, \tau$
Output: $\widehat{L}_\ell^{\{k\}}(0, \Delta)$
$t^{(r)} = \tilde{t}_{\max}^{H_{k\ell}} + b_\ell - \Delta_\ell^A, \quad t^{(l)} = \tilde{t}_{\max}^{H_{k\ell}} - b_\ell + \Delta_\ell^A$
$\widehat{S}_\ell^{\{k\}}(t) = \begin{cases} 1 & t \in [t^{(l)}, t^{(r)}] \\ 0 & \text{otherwise} \end{cases}$        ▷ extended burst
for $i = 1$ to $\left\lceil \frac{\tau - t^{(r)}}{per_\ell} \right\rceil$ do        ▷trace for $t > t^{(r)}$
    $\widehat{S}_\ell^{\{k\}}(t) = \widehat{S}_\ell^{\{k\}}(t) + \nu_\ell\left(t, t^{(r)} + (i-1) \cdot per_\ell\right)$
end for
for $i = 1$ to $\left\lceil \frac{t^{(l)}}{per_\ell} \right\rceil$ do        ▷trace for $t < t^{(l)}$
    $\widehat{S}_\ell^{\{k\}}(t) = \widehat{S}_\ell^{\{k\}}(t) + \nu_\ell\left(t, \Delta_\ell^I + t^{(l)} - i \cdot per_\ell\right)$
end for
$\widehat{L}_\ell^{\{k\}}(0, \Delta) = \int_0^\Delta \widehat{S}_\ell^{\{k\}}(\xi)\, d\xi$ for all $0 \leq \Delta \leq \tau$
```

Lem. 5.7: **(Constant Slope Mode Function)** *Suppose that mode function:*

$$\breve{S}_\ell(t) = \begin{cases} 1 & \tilde{t}^{H_{k\ell}}_{\max} - b_\ell \le t \le \tilde{t}^{H_{k\ell}}_{\max} + b_\ell \\ \delta_\ell & \textit{otherwise} \end{cases} \tag{5.25}$$

with utilization $\delta_\ell = \frac{\Delta^A_\ell}{\Delta^A_\ell + \Delta^I_\ell}$ *leads to temperature* $\breve{T}^*_{k,\ell}(\tau)$ *at time* τ*. When the scheduler is work-conserving,* $\breve{T}^*_{k,\ell}(\tau)$ *is an upper bound on the highest value of temperature* $T_{k,\ell}(\tau)$ *at time* τ*.*

Proof. Rewriting Eq. (5.12) with Eq. (5.13) leads to:

$$T_{k,\ell}(t) = u^i_\ell \cdot \int_0^t H_{k\ell}(t-\xi)\,\mathrm{d}\xi + (u^a_\ell - u^i_\ell) \cdot \int_0^t S_\ell(\xi) \cdot H_{k\ell}(t-\xi)\,\mathrm{d}\xi. \tag{5.26}$$

As we know from Lemma 5.6 that $\widehat{S}(t) = \frac{\mathrm{d}\widehat{L}^{\{k\}}_\ell(0,\Delta)}{\mathrm{d}t}$ leads to $\widehat{T}_{k,\ell}(\tau)$ with $\widehat{T}_{k,\ell}(\tau) \ge T_{k,\ell}(\tau)$, we have to show that $\breve{T}_{k,\ell}(\tau) \ge \widehat{T}_{k,\ell}(\tau)$:

$$\begin{aligned} &\left(\breve{T}_{k,\ell}(\tau) - \widehat{T}_{k,\ell}(\tau)\right) / \left(u^a_\ell - u^i_\ell\right) \\ &= \int_0^\tau \breve{S}_\ell(\xi) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi - \int_0^\tau \widehat{S}_\ell(\xi) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \\ &= \int_0^{\theta^{(l)}} \breve{S}_\ell(\xi) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi - \int_0^{\theta^{(l)}} \widehat{S}_\ell(\xi) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \\ &\quad + \int_{\theta^{(r)}}^\tau \breve{S}_\ell(\xi) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi - \int_{\theta^{(r)}}^\tau \widehat{S}_\ell(\xi) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \end{aligned} \tag{5.27}$$

where we used that $\breve{S}_\ell(t) = \widehat{S}_\ell(t) = 1$ for $t \in [\theta^{(l)}, \theta^{(r)}]$ with $\theta^{(l)} = \tilde{t}^{H_{k\ell}}_{\max} - b_\ell$ and $\theta^{(r)} = \tilde{t}^{H_{k\ell}}_{\max} + b_\ell$. By rewriting the integral from 0 to $\theta^{(l)}$ as a sum, we get:

$$\begin{aligned} &\int_0^{\theta^{(l)}} \left(\breve{S}_\ell(\xi) - \widehat{S}_\ell(\xi)\right) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \\ &\quad = \sum_{i=0}^{\varrho} \left(\int_{\theta^{(l)}-(i+1)\cdot per_\ell}^{\theta^{(l)}-i\cdot per_\ell} \left(\breve{S}_\ell(\xi) - \widehat{S}_\ell(\xi)\right) \cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \right) \end{aligned} \tag{5.28}$$

where ϱ is an integer that is selected so that $per_\ell \cdot (\varrho - 1) \le \theta^{(l)} \le per_\ell \cdot \varrho$. In particular, $\widehat{S}(t) = 0$ for $t \in [\theta^{(l)} - i \cdot per_\ell - \Delta^I_\ell, \theta^{(l)} - i \cdot per_\ell]$, $\widehat{S}(t) = 1$ for $t \in [\theta^{(l)} - (i+1) \cdot per_\ell, \theta^{(l)} - (i+1) \cdot per_\ell + \Delta^A_\ell]$, and $\theta^{(l)} - i \cdot per_\ell - \Delta^I_\ell =$

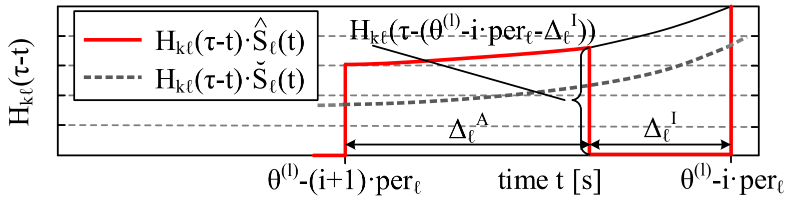


Fig. 5.12: Illustration of the proof of Lemma 5.7. The impulse response function $H_{k\ell}(t)$ is plotted for the interval $[\theta^{(l)} - (i+1)\cdot per_\ell, \theta^{(l)} - i\cdot per_\ell]$.

$\theta^{(l)} - (i+1)\cdot per_\ell + \Delta_\ell^A$, see Fig. 5.12 for an illustration. Then, we find:

$$\begin{aligned}
&\int_{\theta^{(l)}-(i+1)\cdot per_\ell}^{\theta^{(l)}-i\cdot per_\ell} \left(\check{S}_\ell(\xi) - \widehat{S}_\ell(\xi)\right)\cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \\
&\quad = \delta_\ell \cdot \int_{\theta^{(l)}-(i+1)\cdot per_\ell+\Delta_\ell^A}^{\theta^{(l)}-i\cdot per_\ell} H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \\
&\qquad - (1-\delta_\ell)\cdot \int_{\theta^{(l)}-(i+1)\cdot per_\ell}^{\theta^{(l)}-i\cdot per_\ell-\Delta_\ell^I} H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi
\end{aligned} \tag{5.29}$$

where we subtracted the two integrals in the interval $[\theta^{(l)} - (i+1)\cdot per_\ell, \theta^{(l)} - i\cdot per_\ell - \Delta_\ell^I]$. Next, we lower bound the value between $\theta^{(l)} - (i+1)\cdot per_\ell + \Delta_\ell^A$ and $\theta^{(l)} - i\cdot per_\ell$ by means of $H_{k\ell}(\tau - (\theta^{(l)} - (i+1)\cdot per_\ell + \Delta_\ell^A))$, and upper bound the value between $\theta^{(l)} - (i+1)\cdot per_\ell$ and $\theta^{(l)} - i\cdot per_\ell - \Delta_\ell^I$ by means of $H_{k\ell}(\tau - (\theta^{(l)} - i\cdot per_\ell - \Delta_\ell^I)) = H_{k\ell}(\tau - (\theta^{(l)} - (i+1)\cdot per_\ell + \Delta_\ell^A))$, as well:

$$\begin{aligned}
&\int_{\theta^{(l)}-(i+1)\cdot per_\ell}^{\theta^{(l)}-i\cdot per_\ell} (\check{S}_\ell(\xi) - \widehat{S}_\ell(\xi))\cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \\
&\geq \delta_\ell \cdot \Delta_\ell^I \cdot H_{k\ell}(\tau - (\theta^{(l)} - (i+1)\cdot per_\ell + \Delta_\ell^A)) \\
&\quad - (1-\delta_\ell)\cdot \Delta_\ell^A \cdot H_{k\ell}(\tau - (\theta^{(l)} - (i+1)\cdot per_\ell + \Delta_\ell^A)) = 0
\end{aligned} \tag{5.30}$$

where we used the fact that $\delta_\ell = \frac{\Delta_\ell^A}{\Delta_\ell^A + \Delta_\ell^I}$ and $\delta_\ell \cdot \Delta_\ell^I - (1-\delta_\ell)\cdot \Delta_\ell^A = 0$. Similarly, we can show that:

$$\int_{\theta^{(r)}}^{\tau} \check{S}_\ell(\xi)\cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi - \int_{\theta^{(r)}}^{\tau} \widehat{S}_\ell(\xi)\cdot H_{k\ell}(\tau-\xi)\,\mathrm{d}\xi \geq 0 \tag{5.31}$$

and therefore, $\check{T}_{k,\ell}(\tau) - \widehat{T}_{k,\ell}(\tau) \geq 0$.

□

Finally, we present an analytical expression to calculate a non-trivial upper bound on the maximum temperature $\check{T}_k^*(\tau)$ of node k. Then, an upper bound on the worst-case chip temperature can be obtained by calculating the maximum of all individual upper bounds.

Thm. 5.3: **(Upper Bound on the Worst-Case Peak Temperature)** *Suppose that* $T_k(t)$ *is the temperature of node* k *at time instant* t *for a set of workload functions* $\mathbf{R}(s,t)$ *that are bounded by the set of arrival curves* $\boldsymbol{\alpha}$. *Furthermore, let* $u_\ell(t)$ *be the input of node* ℓ *as defined in Eq.* (5.13). *When the scheduler is work-conserving, the following statements hold:*

- *The temperature:*

$$\breve{T}_k^*(\tau) = T_k^{\text{init}}(\tau) + \sum_{\ell=1}^{\eta} \breve{T}_{k,\ell}^*(\tau) \tag{5.32}$$

is an upper bound on the highest temperature of node k *at time* τ, *i.e.,* $\breve{T}_k^*(\tau) \geq T_k(\tau)$, *whereby:*

$$\begin{aligned} \breve{T}_{k,\ell}^*(\tau) &= \left(u_\ell^i + \delta_\ell \cdot \left(u_\ell^a - u_\ell^i\right)\right) \cdot \int_0^\tau H_{k\ell}(t-\xi)\,\mathrm{d}\xi \\ &+ \left(u_\ell^a - u_\ell^i\right) \cdot (1-\delta_\ell) \cdot \int_{\tilde{t}_{\max}^{H_{k\ell}}-b_\ell}^{\tilde{t}_{\max}^{H_{k\ell}}+b_\ell} H_{k\ell}(t-\xi)\,\mathrm{d}\xi, \end{aligned} \tag{5.33}$$

and $\delta_\ell = \frac{\Delta_\ell^A}{\Delta_\ell^A + \Delta_\ell^I}$.

- *In addition, if* $(\mathbf{T}^\infty)^i$ *is the steady-state temperature vector if all nodes are in 'idle' mode,* $\widehat{T}_k^*(\tau) \geq T_k(t)$ *for all* $0 \leq t \leq \tau$ *and any set of feasible workload traces with the same initial temperature vector* $\mathbf{T}^0 \leq (\mathbf{T}^\infty)^i$.

Proof. First, we rewrite Eq. (5.12) with Eq. (5.25) to derive Eq. (5.33). As Lemma 5.7 states that $\breve{T}_{k,\ell}^*(\tau) \geq \breve{T}_{k,\ell}(\tau)$ for all ℓ, and $T_k(t) = T_k^{\text{init}}(t) + \sum_{\ell=1}^{\eta} T_{k,\ell}(t)$, we get $\breve{T}_k^*(\tau) \geq T_k(\tau)$.

The second item is a simple consequence of Lemma 5.6. As $\breve{T}_k^*(\tau) \geq \widehat{T}_k^*(\tau)$, $\breve{T}_k^*(\tau) \geq T_k(t)$ for all $t \leq \tau$. □

Three different methods to calculate an upper bound on the maximum temperature have been presented in this section. The first method calculates the critical accumulated computing time with Algorithm 5.1 leading to the worst-case peak temperature T_k^* of node k. The second method calculates the accumulated computing time according to Eq. (5.24) leading to $\widehat{T}_k^*$, and the last method calculates an upper bound on the maximum temperature $\breve{T}_k^*$ of node k with the analytical expression defined in Eq. (5.32). The relation between these three different bounds on the maximum temperature is as follows:

$$\breve{T}_k^*(\tau) \geq \widehat{T}_k^*(\tau) \geq T_k^*(\tau). \tag{5.34}$$

5.6 Automatic Generation and Calibration of Analysis Models

Automation is the key for fast design space exploration. Design decisions are taken on the basis of comparing different candidate mappings and how well they perform with respect to both timing and temperature. Integrating formal timing and thermal analysis models in the system-level design space exploration loop requires both automatic generation of analysis models and automatic calibration of these models with performance data reflecting the execution of the system on the target platform. For today's multi- and many-core platforms, the manual generation of such analysis models and the individual calibration of all components with platform-dependent data is a major effort, which would slow down the entire design cycle and would be an actual source of errors.

In this section, we argue that for the considered class of systems, the generation of formal analysis models can be fully automated and that the thermal and timing analysis models can be parameterized from model calibration based on a set of benchmark mappings that are executed prior to design space exploration. In particular, we focus on the generation and calibration of the formal thermal analysis model proposed in Section 5.4. The generation and calibration of timing analysis models is detailed in [HHB+12].

5.6.1 Generation of Analysis Models

The generation of a formal thermal analysis model similar to the one introduced in Section 5.4 consists of translating system specification $S = \langle p, \mathcal{A}, \langle \Gamma, \sigma \rangle \rangle$ into the thermal analysis model $\Pi = \langle \boldsymbol{\alpha}, \Theta \rangle$. Every arrival curve $\alpha_\ell \in \boldsymbol{\alpha}$ upper bounds the total cumulative workload of a core c_ℓ and complies to the standard event model described in Section 5.4.1. Θ refers to the thermal model of the platform, which is characterized by the impulse responses (represented by matrix $\mathbf{H}$) and the power consumption vector $\boldsymbol{\mathcal{P}}$. As the processing components are supposed to be work-conserving, the resource availability of core c_ℓ is implicitly abstracted by the service curve $\beta_\ell(\Delta) = \Delta$ for all intervals of length $\Delta \geq 0$.

The two-step procedure outlined in Fig. 5.13 is used to calculate the set of arrival curves $\boldsymbol{\alpha}$. In a first step, the MPA model $\mathcal{M} = \langle \mathcal{V}, E \rangle$ of system $S = \langle p, \mathcal{A}, \langle \Gamma, \sigma \rangle \rangle$ is generated. For each process $v \in V$ of process network $p = \langle V, Q \rangle$, an MPA Greedy Processing Component (GPC) is instantiated and for every FIFO channel $q \in Q$, an event stream is instantiated between the corresponding actors, by means of an MPA arrival curve. Afterwards, binding and scheduling of processes are addressed by connecting GPCs with corresponding resource streams. For example, preemptive fixed

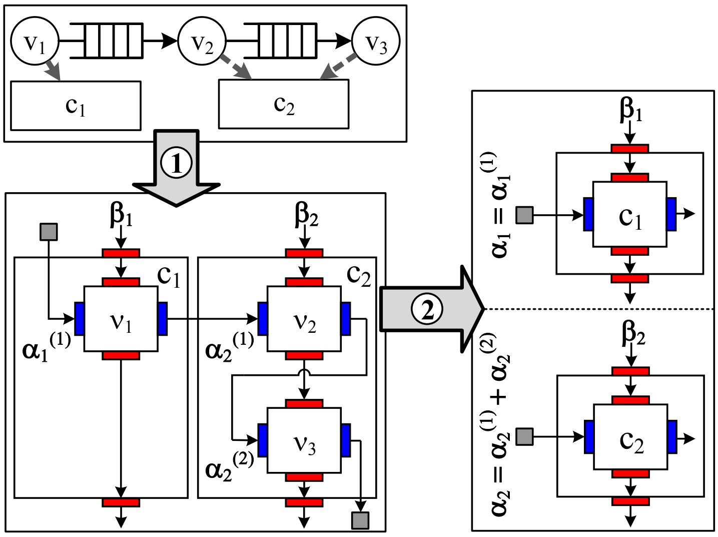


Fig. 5.13: Steps to calculate the computational demand of two processing cores. In particular, three processes v_1, v_2, and v_3 are assigned to two cores c_1 and c_2.

priority scheduling can be modeled by chaining the GPCs in order of their priority, and assigning full resource availability to the first GPC of the chain. In Fig. 5.13, full resource availability is assigned to the GPC corresponding to v_2, and the remaining resource availability is connected to the GPC corresponding to v_3.

In the second step, the set of arrival curves $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_{|C|}]'$ is calculated from the previously generated MPA model $\mathcal{M} = \langle \mathcal{V}, E \rangle$. Suppose that actor $\mathcal{V}_j$ represents process v_j and has the arrival curve $\alpha^{(j)}$. Then, the arrival curve α_ℓ of core c_ℓ is:

$$\alpha_\ell(\Delta) = \sum_{j=1}^{|V|} \Gamma(v_j, c_\ell) \cdot \alpha^{(j)}(\Delta). \tag{5.35}$$

In case that the arrival curve α_ℓ is not complying to the standard event model anymore, it is necessary to convert the arrival curve into the standard event model with the method presented in [KHET07]. As the method upper bounds the original arrival curve, the proposed thermal analysis model still leads to a safe bound on the worst-case chip temperature. Figure 5.14 illustrates this step by means of a processing core that has three processes assigned.

The generation of the thermal model Θ follows directly from the platform specification. First, the coefficients $\boldsymbol{\phi}$, which reflect the temperature-dependency of the leakage power, are calculated by linearizing the power

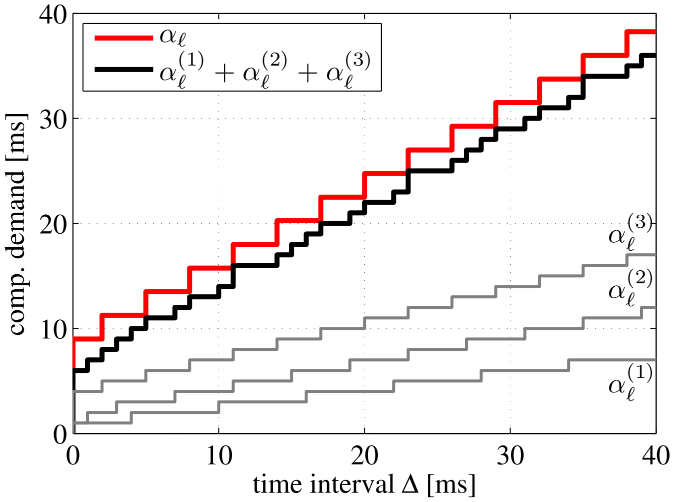


Fig. 5.14: Calculation of the arrival curve α_ℓ for processing core c_ℓ that has three processes assigned. $\alpha_\ell^{(1)}$, $\alpha_\ell^{(2)}$, and $\alpha_\ell^{(3)}$ are the arrival curves of the corresponding processes.

model described in [SSS⁺04]. Then, the power consumption vector $\boldsymbol{\mathcal{P}}$ is calculated as follows:

$$\boldsymbol{\mathcal{P}}(t) = \boldsymbol{\phi} \cdot \left(\mathbf{T}(t) - \mathbf{T}^{\text{base}}\right) + \boldsymbol{\mathcal{P}}_{\text{leakage}} + \boldsymbol{\mathcal{P}}_{\text{dyn}}(t) = \boldsymbol{\phi} \cdot \mathbf{T}(t) + \boldsymbol{\psi}(t) \qquad (5.36)$$

where $\mathbf{T}^{\text{base}}$ is the temperature vector for which the calibration parameters are obtained. The set of impulse responses is finally calculated with:

$$\mathbf{H}(t) = e^{\mathbf{A}\cdot t} \cdot \mathbf{B} = e^{-\mathcal{C}^{-1}\cdot(\mathbf{G}+\mathbf{K}-\boldsymbol{\phi})\cdot t} \cdot \mathcal{C}^{-1}. \qquad (5.37)$$

5.6.2 Calibration of Analysis Models

During model generation, the timing behavior of a system is abstracted by arrival and service curves. The thermal behavior is abstracted by a temperature-dependent power consumption model and a thermal *RC* network. In order to obtain the required parameters for this abstraction, the application is calibrated prior to design space exploration with data corresponding to the target platform. Table 5.2 summarizes the required parameters and categorizes them into two subsets, namely timing parameters ψ_{timing} and thermal parameters ψ_{thermal}. While timing analysis only depends upon the timing parameters ψ_{timing}, both subsets are required for the calculation of the worst-case chip temperature. Timing parameters include the worst-case and best-case execution times of a process and the characteristics of the channels. The thermal parameters include the power consumption and the thermal configuration of the platform.

Tab. 5.2: Model parameters required to parameterize the thermal and timing analysis models. The parameters are categorized into two subsets, namely timing parameters ψ_{timing} and thermal parameters ψ_{thermal}.

entity	**parameter**	**unit**	**category**
process v	worst-case execution time / best-case execution time	s / iteration	time
queue q	min. / max. token size	bytes / access	time
	min. / max. write and read rate	1	time
core c	clock frequency f	cycles / s	time
	dynamic power consumption $\overline{\mathcal{P}}_{\text{dyn}}(c)$	W	thermal
	leakage power consumption $\overline{\mathcal{P}}_{\text{leakage}}(c)$	W	thermal
architecture $\mathcal{A}$	capacitance matrix $\boldsymbol{C}$	J/K	thermal
	conductance matrix $\mathbf{G}$ / ground conductance matrix $\mathbf{K}$	W/K	thermal

Both the execution time of processes and the power consumption of the cores are associated with an uncertainty. Formal methods and tools [WEE⁺08] are actually required to calculate hard bounds on the execution time and the power consumption. However, such strict formal methods have the disadvantage that the overall modeling effort drastically increases with the complexity of the system. An alternative for model calibration is simulation or execution on real hardware platforms. Although safe bounds cannot be guaranteed unless exhaustive test patterns are applied, they are often the only practical possibility for model calibration. Compared to real hardware platforms, simulation has the advantage that virtual platforms are often earlier available and allow non-intrusive tracking of the execution. Therefore, in Section 5.8, the prototype implementation of the mapping optimization framework uses simulation on a cycle-accurate virtual platform to determine worst-case execution times and power consumptions.

Time-independent parameters, i.e., the token consumption behavior of the channels, are determined by monitoring the READ and WRITE calls during functional simulation. Time-dependent parameters, namely the best-case and worst-case execution times of processes are obtained by monitoring the system in a cycle-accurate simulator. After tracking all context switches, the individual execution times of processes are calculated by means of a log-file analysis, where the collected information is first decoded and then stored in a database.

In order to predict the worst-case chip temperature, thermal parameters, which provide safe bounds on power dissipation, have to be determined. As described in Table 5.2, the set of thermal parameters ψ_{thermal} is composed of two assignments:

$$\psi_{\text{thermal}} : \begin{cases} c \mapsto \left\langle f, \overline{\mathcal{P}}_{\text{leakage}}(c), \overline{\mathcal{P}}_{\text{dyn}}(c) \right\rangle & \forall c \in C \\ \mathcal{A} \mapsto \langle \boldsymbol{C}, \mathbf{G}, \mathbf{K} \rangle. \end{cases} \tag{5.38}$$

A possible method to determine the maximum power consumption of a processing core is to sum up the power dissipations of all individual units, according to data provided by hardware designers in data sheets. However, it is unlikely that all units are simultaneously active so that the obtained power numbers would drastically outperform the actual power consumption. Therefore, we use, again, a simulation-based approach to obtain better estimates of the power consumption.

The power parameters of each processing core are obtained by monitoring the dynamic and leakage power dissipation separately in a cycle-accurate simulator. As the thermal model assumes time-independent leakage power, $\overline{\mathcal{P}}_{\text{leakage}}(c)$ is set to the maximum value of the leakage power that is observed for core c during execution. A multi-step procedure is required to determine the dynamic power consumption. First, the transient dynamic power consumption of each core is recorded as power traces. Second, the power traces are divided into individual processes so that the average dynamic power consumption $\overline{\mathcal{P}}^{v,i}_{\text{dyn}}$ of process v in iteration i is determined as:

$$\overline{\mathcal{P}}^{v,i}_{\text{dyn}} = \frac{1}{t^{v,i}_{\text{end}} - t^{v,i}_{\text{start}}} \int_{t^{v,i}_{\text{start}}}^{t^{v,i}_{\text{end}}} \mathcal{P}^{v}_{\text{dyn}}(t)\, \mathrm{d}t \tag{5.39}$$

where $t^{v,i}_{\text{start}}$ and $t^{v,i}_{\text{end}}$ are the start time and end time of iteration i, respectively. Then, we use the maximum average value of $\mathcal{P}_{\text{dyn}}(t)$ as dynamic power consumption $\overline{\mathcal{P}}_{\text{dyn}}(c)$:

$$\overline{\mathcal{P}}_{\text{dyn}}(c) = \max_{\forall v \text{ assigned to } c} \left(\max_{\forall i} \left(\overline{\mathcal{P}}^{v,i}_{\text{dyn}} \right) \right). \tag{5.40}$$

The thermal matrices $\mathbf{G}$, $\mathbf{K}$, and $\boldsymbol{C}$ are computed by means of the method described in [SSS+04]. The parameters required for this calculation, in particular the three-dimensional floorplan and the thermal configuration of the chip, are known from the architecture specification.

In our mapping optimization framework, model calibration is performed just once in the design flow, namely before design space exploration. As all functional parameters are mapping-independent, they only have to be obtained once for all candidate mappings. The same applies to the thermal configuration of the platform, i.e., the $\mathbf{G}$, $\mathbf{K}$, and $\boldsymbol{C}$ matrices. All other parameters obtained by cycle-accurate simulation vary with each candidate mapping, but can be estimated by a set of benchmark mappings that covers all possible configurations. In the following experiments, we use the MPARM [BBB+05] virtual platform for cycle-accurate simulation. As the MPARM virtual platform is composed of homogeneous cores and uses direct memory access controllers to reduce the interference between computation and communication, the number of benchmark mappings can be reduced to one.

5.7 Minimizing the Peak Temperature

So far, we have seen a method to calculate the worst-case chip temperature and a method to generate the required formal thermal analysis model automatically. Next, we apply these methods to calculate a process-to-core assignment that minimizes the worst-case chip temperature and guarantees that all real-time deadlines are met.

5.7.1 Peak Temperature Minimization Problem

In the following, we state the considered optimization problem aiming to reduce the worst-case chip temperature. However, before being able to state the optimization problem, we introduce the concept of a demand bound function [BMR90] that is later used to calculate the maximum computational demand in any time interval.

Demand Bound Function

The demand bound function [BMR90] models the maximum resource demand of a process. More formally, the demand bound function $\mathrm{dbf}_{v_j}(\Delta)$ of process v_j upper bounds the maximum accumulated computational demand of all events that arrive and have deadline in any interval of length Δ. Formally, the demand bound function $\mathrm{dbf}_{v_j}(\Delta)$ is defined by:

$$\mathrm{dbf}_{v_j}(\Delta) = \alpha_j(\Delta - D_j) \quad \forall \Delta \geq 0. \tag{5.41}$$

The demand bound function $\mathrm{dbf}_{c_\ell}(\Delta)$ of core c_ℓ depends on the scheduling algorithm. For example, when an EDF scheduler is used to arbitrate between events of different processes assigned to the same core, the demand bound function $\mathrm{dbf}_{c_\ell}(\Delta)$ is [BMR90]:

$$\mathrm{dbf}_{c_\ell}(\Delta) = \sum_{j=1}^{|V|} \Gamma(v_j, c_\ell) \cdot \mathrm{dbf}_{v_j}(\Delta). \tag{5.42}$$

Optimization Problem

Now, we are able to formulate the considered optimization problem:

> *Given a process network $p = \langle V, Q \rangle$ that is mapped onto a many-core SoC. Then, the goal is to select a static assignment of processes to cores such that all deadlines are met and the worst-case chip temperature T_S^* is minimized.*

In other words, the objective of the optimization problem is to reduce the worst-case chip temperature:

$$\text{minimize } T_S^* = \max\left(T_1^*, \ldots, T_\eta^*\right) \tag{5.43}$$

where T_k^* is the worst-case peak temperature of node k and η is the number of nodes of the equivalent thermal RC circuit.

In order to guarantee that the real-time deadlines of all events are met, we have to make sure that the cumulated number of available computing resources is in no time interval Δ smaller than the maximum resource demand. To this end, we use the concept of the demand bound function. Then, the schedulability test can be written as follows:

$$\mathrm{dbf}_{c_\ell}(\Delta) \leq \Delta \quad \forall \Delta \geq 0 \text{ and } c_\ell \in C. \tag{5.44}$$

Finally, we have to make sure that each process is assigned to only one core:

$$\sum_{c_\ell \in C} \Gamma(v, c_\ell) = 1 \quad \forall v \in V. \tag{5.45}$$

The previously stated thermal optimization problem can be solved with a wide variety of optimization techniques including the techniques discussed in Chapter 2. For illustration purpose, we will use simulated annealing [KGV83] to solve the optimization problem in the following case studies.

5.7.2 Temperature Reduction by Voltage Scaling

In addition, the worst-case chip temperature can be reduced by assigning each core its optimal frequency, i.e., the minimum operation frequency so that no real-time deadlines are missed. In the following, we extend both the system and thermal analysis model and formulate the optimization problem to make use of voltage and frequency scaling to reduce the power consumption and thus the worst-case chip temperature.

Each core c_ℓ has its own clock domain and executes at a static frequency f_ℓ with $0 \leq f_\ell \leq f_\ell^{\max}$. We suppose that the runtime of process v scales linearly with the operation frequency. Thus, the total accumulated workload of c_ℓ is upper bounded by the arrival curve:

$$\alpha_\ell(\Delta) = \frac{f_\ell^{\max}}{f_\ell} \cdot \sum_{j=1}^{|V|} \Gamma(v_j, c_\ell) \cdot \alpha_j(\Delta). \tag{5.46}$$

Furthermore, we assume that the dynamic power consumption of core c_ℓ grows quadratically with its supply voltage ν_ℓ and linearly with its operation frequency f_ℓ [RCN08]:

$$\mathcal{P}_{\ell,\mathrm{dyn}}(t) \propto \nu_\ell^2 \cdot f_\ell \cdot S_\ell(t). \tag{5.47}$$

Similar to [MMA⁺07], we suppose that the square of the supply voltage scales linearly with the operation frequency even though the results of this section also hold for any other monotonic relation between supply voltage and frequency. Then, the total power consumption is given by:

$$\boldsymbol{\mathcal{P}}(t) = \boldsymbol{\phi} \cdot \mathbf{T}(t) + \boldsymbol{\rho} \cdot \mathrm{diag}(\mathbf{f})^3 \cdot \mathbf{S}(t) + \boldsymbol{\omega} \tag{5.48}$$

with diagonal matrix diag($\mathbf{f}$) of vector $\mathbf{f}$ and constant diagonal matrices $\boldsymbol{\rho}$ and $\boldsymbol{\omega}$. As the operation frequency is statically assigned at design time, the thermal analysis method proposed in Section 5.5 still provides an upper bound on the maximum temperature.

In order to calculate the minimum operation frequency so that no real-time deadlines are missed, we rewrite the demand bound function $\mathrm{dbf}_{c_\ell}(\Delta)$ with the scaled operation frequency f_ℓ:

$$\mathrm{dbf}_{c_\ell}(\Delta) = \frac{f_\ell^{\max}}{f_\ell} \cdot \sum_{j=1}^{|V|} \Gamma(v_j, c_\ell) \cdot \mathrm{dbf}_{v_j,c_\ell}(\Delta). \tag{5.49}$$

Finally, rewriting Eq. (5.44) with the above expression for the demand bound function results in the following expression for the minimum operation frequency for a core that uses an EDF scheduler to arbitrate between events of different processes:

$$f_\ell = \sup_{\Delta \geq 0} \left\{ f_\ell^{\max} \cdot \frac{\sum_{j=1}^{|V|} \Gamma(v_j, c_\ell) \cdot \mathrm{dbf}_{v_j,c_\ell}(\Delta)}{\Delta} \right\}. \tag{5.50}$$

5.8 Evaluation

In this section, a prototype implementation of the proposed system-level mapping optimization framework is used to explore different system designs with respect to both worst-case latency and temperature. In the first case study, the viability of the proposed thermal analysis methods is discussed by comparing the maximum temperatures obtained with four different evaluation methods. Then, in the second case study, we evaluate the time to perform design space exploration and the quality of the obtained temperature bounds. Finally, Pareto-optimal candidate mappings are identified for two benchmark applications and the effect of frequency and voltage scaling on the worst-case chip temperature is evaluated. A mapping is considered to be Pareto-optimal if, for a given performance, no other mapping has a lower worst-case chip temperature.

5.8.1 Experimental Setup

The target platform is the MPARM [BBB+05] cycle-accurate simulator that emulates an ARM-based MPSoC. The reconfigurable platform is composed of a variable number of identical 32-bit ARM 7 cores and shared memories, which are connected by a shared bus. The communication channels are implicitly assumed to be mapped onto the scratchpad of the core of the sender process. If not stated otherwise, thermal management methods have been deactivated, but it is assumed that the core can switch to a power-saving mode if no process is executed. If not specified otherwise, fixed priority preemptive scheduling is used on all cores while a TDMA policy is employed on the bus.

In order to execute DAL applications on top of the MPARM virtual platform, the DOL software synthesis back-end for the MPARM simulator [HHBT09] has been ported to the DAL design flow. The proposed thermal analysis methods have been integrated into the DAL design flow by extending the MPA framework [WTVL06] with the ability to calculate the worst-case chip temperature by the methods proposed in Section 5.5.

Automated model calibration is performed based on timing and power numbers extracted from the MPARM virtual platform and thermal parameters extracted from HotSpot [HGV+06]. The considered power model of the MPARM virtual platform is detailed in [Sch11] and the considered thermal configuration of HotSpot is summarized in Table 5.3. The floorplans used in all case studies are based on the examples given in [ADVP+07], but adjusted to the MPARM virtual platform and the number of cores of the considered target architecture.

All experiments have been performed on a 3.20 GHz Intel Pentium D machine with 2 GB of RAM.

Tab. 5.3: Thermal configuration of HotSpot [HGV+06].

parameter	symbol	value
silicon thermal conductance [W/(m · K)]	k_{chip}	150
silicon specific heat [J/(m^3 · K)]	p_{chip}	$1.75 \cdot 10^6$
thickness of the chip [mm]	t_{chip}	3.5
convection resistance [K/W]	r_{convec}	2
heatsink width [m]	s_{sink}	0.017
heatsink thickness [mm]	t_{sink}	0.01
heatsink thermal conductance [W/(m · K)]	k_{sink}	400
heatsink specific heat [J/(m^3 · K)]	p_{sink}	$3.55 \cdot 10^6$
thickness of the interface material [mm]	$t_{\text{interface}}$	0.04
ambient temperature [K]	T_{amb}	300

5.8.2 Benchmark Applications

In this section, we analyze the timing and temperature behavior of four benchmark applications:

Producer-Consumer (P-C)

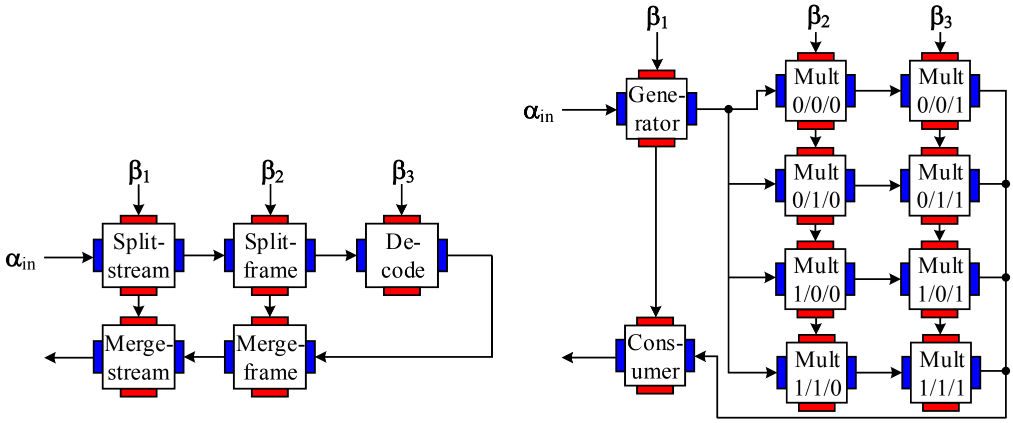
The P-C benchmark is a simple application that consists of five pipelined processes. The 'producer' generates a stream of floating-point numbers, which are passed to the first 'worker' process. After computing a few arithmetic operations, each 'worker' process forwards the floating-point number to the next 'worker' process until the 'consumer' receives it.

MJPEG Decoder

MJPEG [Wal92] is a video compression format in which each video frame is compressed as a JPEG image. The MPA model of the considered configuration is outlined in Fig. 5.15a. The decoder's input, abstracted by the arrival curve α_{in}, is a video stream. The first actor splits the video stream into individual frames. Then, the second actor splits the frames into macroblocks to decode each macroblock separately. Afterwards, the decoded macroblocks are stitched together into a frame and finally into a stream.

Fast Fourier Transform (FFT)

In order to compute an 8-point FFT, a distributed log(8)-stage butterfly network [WC93] has been implemented. Each stage is composed of four multiply-subtract-add modules so that the application can be split up into twelve processes, each computing a multiply-subtract-add module.



(a) MJPEG decoder. **(b)** Matrix multiplication.

Fig. 5.15: Component models of the considered benchmark applications. For the sake of simplicity, the component model of the bus is not shown. α_{in} abstracts the system's input stream. β_1, β_2, and β_3 abstract the available resources of core c_1, c_2, and c_3, respectively.

Matrix Multiplication

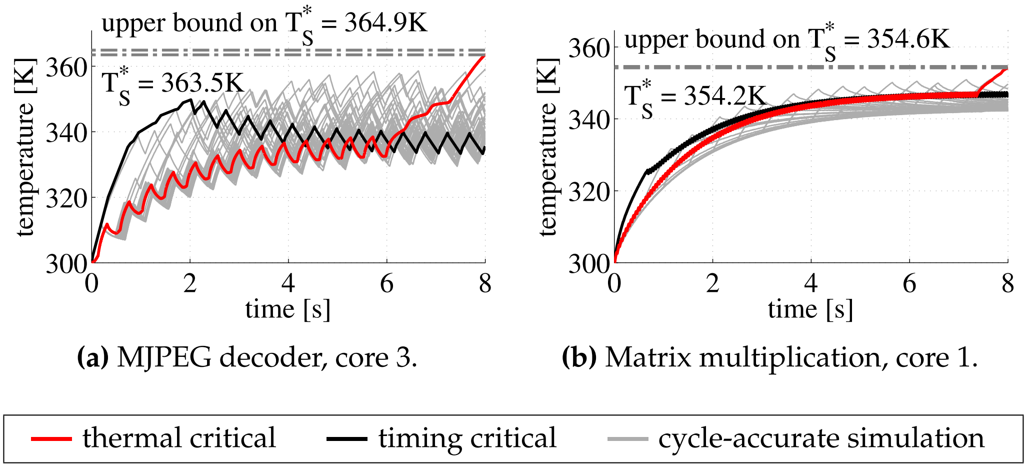
The distributed matrix multiplication application computes the product of two $N \times N$ matrices by subdividing the matrix product into single multiplications and additions. This way, the application is split up into single processes, each performing a multiplication followed by an addition. To feed the structure and to retrieve the result, an input generator and an output consumer are added. Figure 5.15b shows the MPA model of the considered configuration for $N = 2$.

5.8.3 Peak Temperature Analysis

First, we study the viability of the thermal analysis methods proposed in Section 5.5. To this end, we compare the worst-case chip temperature that is calculated by the proposed thermal analysis methods with the maximum temperature that is obtained by two widely used temperature evaluation methods. Afterwards, we apply the proposed thermal analysis methods to explore the temperature distribution on a 25-core processor.

Transient Temperature Evolution

We start by comparing the transient temperature evolution and the resulting maximum temperature for different evaluation methods if the target platform is configured to consist of three processing cores. In particular, we compare the worst-case chip temperature obtained by using the formula stated in Eq. (5.32), the temperature evolution of the thermal



(a) MJPEG decoder, core 3.

(b) Matrix multiplication, core 1.

Fig. 5.16: Temperature evolution of the node corresponding to the core with the maximum temperature when the system is processing the MJPEG decoder application or the matrix multiplication application.

critical instance, the temperature evolution of the timing critical instance, and the temperature evolution of 40 workload traces simulated on a cycle-accurate simulation tool-chain based on the MPARM virtual platform and HotSpot. The thermal critical instance is the trace that leads to the worst-case chip temperature and the timing critical instance is the trace that releases the workload as early as possible. In order to stress the system, the 40 input traces executed on the cycle-accurate simulation tool-chain cover the whole range of possible input streams. In particular, among the 40 traces, there are input streams where the number of events arriving concurrently equals the maximum length of the input stream's burst.

Figure 5.16 presents the results of the transient temperature simulation in the interval $[0, 8\,]s$ when the observation time τ is set to eight seconds. Even though we only show the temperature evaluation of two benchmark applications, the findings for the other benchmark applications show similar trends.

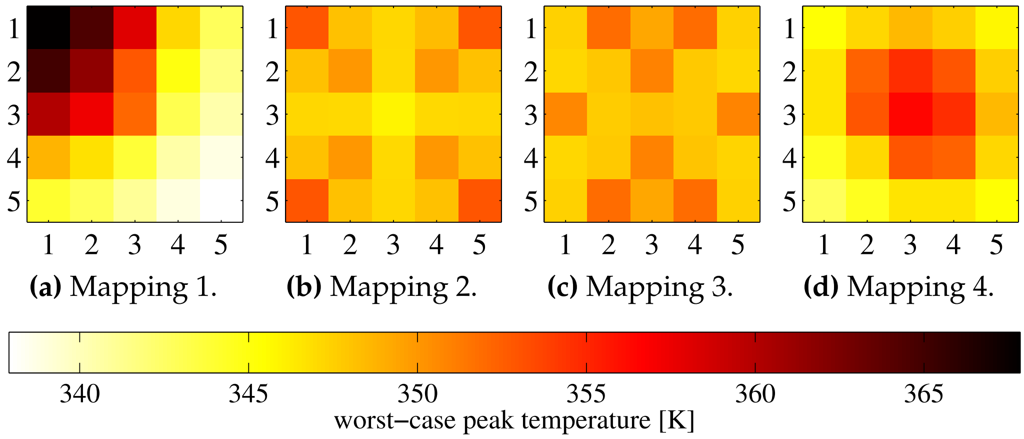
The maximum temperature caused by the timing critical instance is 349.8 K for the MJPEG decoder application and 347.3 K for the matrix multiplication application. As it releases the workload as early as possible, the timing critical instance heats up the system faster than any other trace at the beginning, but the system cools down at a later point in time. All other traces that are recorded on the cycle-accurate simulation tool-chain heat up the chip the most when multiple events arrive concurrently. In particular, the average maximum temperature caused by these workload traces is 352.4 K and 346.1 K, and the highest maximum temperature is 360.1 K and 351.1 K for the MJPEG decoder application and the matrix multiplication application, respectively. The thermal critical instance places all bursts close to the observation time τ so that at time $\tau = 8\,\mathrm{s}$, a maximum temperature of 363.5 K is observed for the MJPEG decoder application. Furthermore, Eq. (5.32) leads to an upper bound on the worst-case chip temperature of 364.9 K for the MJPEG decoder application. The worst-case chip temperature of the matrix multiplication application is 354.2 K if the thermal critical workload is simulated and Eq. (5.32) leads to an upper bound on the worst-case chip temperature of 354.6 K.

Workload traces, which have an input stream where the number of concurrently arriving events equals the maximum length of the input stream's burst, lead to a high maximum temperature that might be only a few degrees below the worst-case chip temperature. The difference between the maximum temperature of such workload traces and the worst-case chip temperature may be caused due to several reasons. First, the power consumption depends on various impact factors like cache misses and bus congestions. Second, as shown in Section 5.5, the worst-case chip temperature occurs when all cores simultaneously process a specific pattern, which might be different from a bursty input stream. The small difference between calculating the worst-case chip temperature using the formula stated in Eq. (5.32) and simulating the thermal critical instance is mainly attributed to the fact that the heat transfer among neighboring cores is much smaller than the effect of self-heating. For self-heating, the length of the burst is the same for both methods as $\tilde{t}_{\max}^{H_{k\ell}}$ (see Section 5.4.2) is equal to the observation time τ.

The average durations for simulating the transient temperature evolutions are shown in Table 5.4. Simulating the temperature evolution on the cycle-accurate simulation tool-chain is two orders of magnitude slower than calculating the thermal critical instance and four orders of magnitude slower than calculating an upper bound on the maximum temperature using the formula stated in Eq. (5.32).

Tab. 5.4: Durations for simulating the transient temperature evolution for different evaluation methods and calculating an upper bound on the worst-case chip temperature using the formula stated in Eq. (5.32).

	P-C	**MJPEG**	**FFT**	**Matrix**
upper bound according to Eq. (5.32)	0.43 s	0.10 s	0.56 s	0.45 s
thermal critical instance	84.8 s	94.3 s	83.0 s	90.5 s
timing critical instance	1.8 s	2.2 s	2.2 s	2.2 s
cycle-accurate simulator	24’876 s	22’428 s	22’873 s	23’372 s



(a) Mapping 1. **(b)** Mapping 2. **(c)** Mapping 3. **(d)** Mapping 4.

Fig. 5.17: Worst-case peak temperature distribution for a 25-core processor when executing an MJPEG decoder application. The cores are arranged in a grid with five rows.

Temperature Distribution on a 25-Core Processor

Next, we consider a many-core platform with 25 cores executing an MJPEG decoder with 10 processes. The cores are arranged in a grid with five rows and the corresponding thermal model has order[3] 112. We will show that the temperature distribution, and thereby the worst-case chip temperature of the system, is affected by the process-to-core assignment.

Figure 5.17 shows the worst-case chip temperature distribution of the system for four different mappings. Motivated by the previous results, we calculate the worst-case chip temperature using Eq. (5.32). In Fig. 5.17a, the processes are mapped onto cores situated in the left top corner of the chip. Next, in Fig. 5.17b, the processes are distributed among cores in all four corners. In Fig. 5.17c, the processes are distributed all over the chip, and finally, in Fig. 5.17d, the processes are mapped onto cores in the middle of the chip. The highest peak temperature occurs in Fig. 5.17a and the lowest one in Fig. 5.17c. The difference between their worst-case chip temperatures is of about 16 K. This shows that the worst-case chip temperature can be reduced by spreading the workload over the chip. In this case, intermediate cores with no workload act like a passive cooling system and keep hot spots separated.

5.8.4 Efficiency and Quality

After discussing the viability of the thermal analysis methods, we integrate them into the mapping optimization framework and measure its

---

[3] The number of nodes of the thermal model is four times the number of cores plus 12 additional nodes whereby the factor four comes from the fact that the thermal model has four vertical layers. See Section 5.4.2 for more details.

efficiency and quality. To this end, we first obtain the model parameters from executing the application with a benchmark mapping. Afterwards, during design space exploration, these model parameters are used to analyze different candidate mappings, whereby the worst-case chip temperature is obtained by simulating the thermal critical instance. To measure the efficiency, we compare the duration of analysis model generation, calibration, and evaluation. In order to evaluate the quality of the obtained results, the worst-case chip temperature obtained by simulating the thermal critical instance is compared to the maximum temperature observed on a ten seconds system simulation using the previously described cycle-accurate simulation tool-chain, and the maximum steady-state temperature calculated out of the average power consumption [SSS$^+$04]. Both methods correspond to typical state-of-the-art thermal evaluation methods that are used in thermal-aware task allocation and scheduling framework [XH06, CDH08]. Again, we are targeting the MPARM virtual platform with three cores.

The durations of analysis model generation, calibration, and evaluation are listed in Table 5.5. First, we note that calibration is two to three orders of magnitude slower than model generation and evaluation. Out of the three steps to perform model calibration, cycle-accurate simulation on MPARM is the most time consuming step. The duration of the log-file analysis step mainly depends on the number of context switches, which in turn depends on the number of processes. As a new log entry is created for every context switch, the duration of the log-file analysis is long for the FFT application, where many dependent processes are concurrently executed. Furthermore, both the duration of model calibration and the

Tab. 5.5: Duration of analysis model generation, calibration, and evaluation. The row “synthesis” includes the time for functional simulation and the values reported in row “simulation” refer to the simulation on the MPARM virtual platform.

		P-C	**MJPEG**	**FFT**	**Matrix**
model calibration (one-time effort)	synthesis	37 s	58 s	47 s	39 s
	simulation	24′709 s	22′504 s	22′752 s	23′510 s
	log-file analysis	114 s	114 s	1847 s	292 s
	overall time for one mapping	24′861 s	22675 s	24′646 s	23′842 s
design space exploration	model generation	2 s	3 s	2 s	2 s
	model evaluation	96 s	132 s	96 s	119 s
	overall time for one mapping	98 s	135 s	98 s	121 s

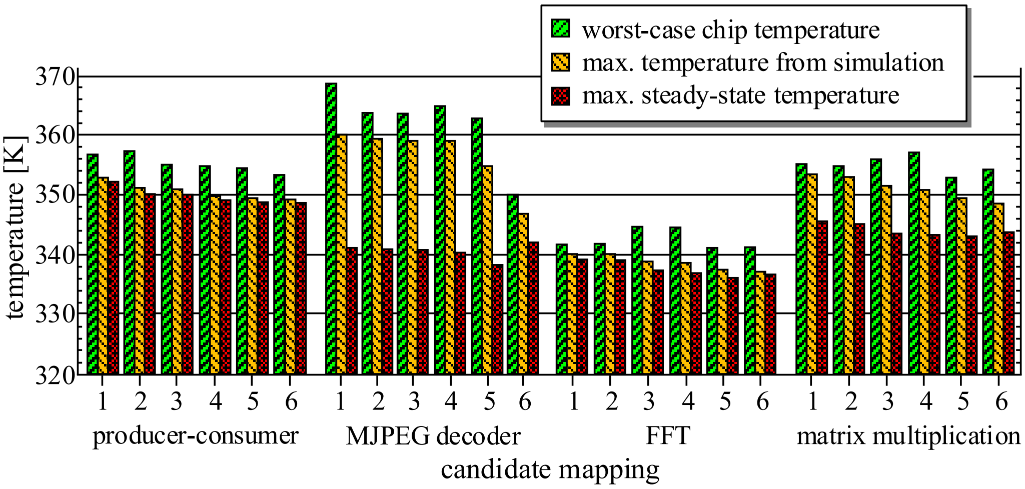


Fig. 5.18: Comparison of the worst-case chip temperature obtained by simulating the thermal critical instance, the maximum temperature observed on a ten seconds system simulation, and the maximum steady-state temperature for six candidate mappings per benchmark application.

accuracy of the obtained results are affected by the length of the execution trace. In general, longer execution traces increase the calibration time but result in better calibration data with respect to worst-case execution time and average power consumption.

In Fig. 5.18, the worst-case chip temperature obtained by simulating the thermal critical instance is compared with the maximum temperature observed on a ten seconds system simulation using the cycle-accurate simulation tool-chain and with the maximum steady-state temperature. In total, we evaluated six randomly selected candidate mappings per benchmark application. The values are ordered by the maximum temperature observed by the system simulation, in descending order.

First, we note that the worst-case chip temperature always upper bounds the maximum temperature obtained from cycle-accurate simulation. As cycle-accurate simulation is several orders of magnitude slower than formal thermal analysis, this confirms our approach to include formal worst-case analysis models in the design space exploration loop and to use cycle-accurate simulation only for model calibration. Second, one can draw the conclusion that the steady-state temperature is no indicator for a low worst-case chip temperature. Third, we note that a mapping candidate with a lower maximum temperature from simulation does not necessary have a lower worst-case chip temperature. On the one hand, the maximum temperature of the cycle-accurate simulation underestimates the worst-case chip temperature due to the infeasibility of an exhaustive simulation of all system configurations. On the other hand, the proposed thermal analysis method does not lead to a tight bound on the maximum temperature of the system, due to several reasons:

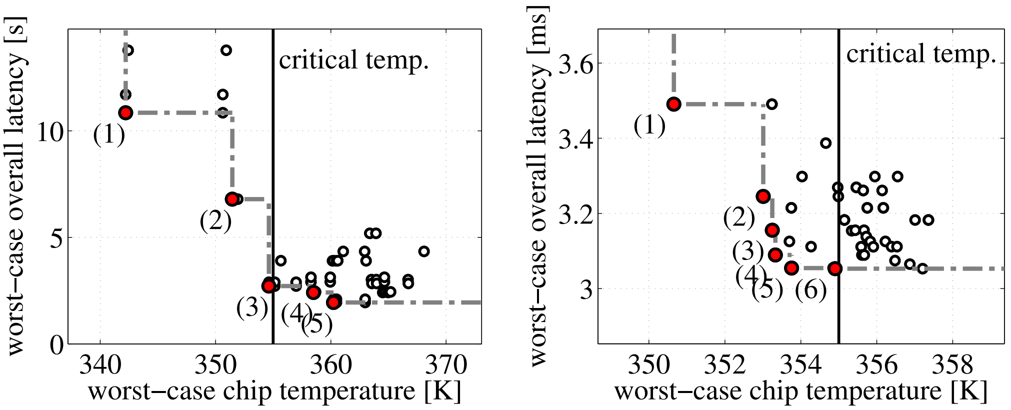
- The parameters modeling the power consumption are the maximum average power consumption per iteration, thus they overestimate the actual power consumption.
- The workload curves of the individual cores are not independent of each other as the component model results from a single process network.
- As not all arrival curves comply with the standard event model, the original arrival curves have to be upper bounded.

Unlike simulation, the proposed thermal analysis method calculates a safe bound on the maximum temperature of a candidate mapping. The difference between the worst-case chip temperature and the maximum temperature of the system simulation is the worst possible inaccuracy of the formal worst-case analysis method. Nonetheless, due to the over-approximation of a tight bound on the maximum temperature of the system, another candidate mapping might actually have a lower maximum temperature. To avoid the selection of a wrong candidate mapping, the designer might keep more than one mapping candidate after the first design space exploration, and reevaluates these candidates with either a higher accuracy or a cycle-accurate simulator.

In summary, calculating the worst-case chip temperature by means of a formal analysis model is desirable. First, identifying the worst-case chip temperature is notably important in real-time systems. The functional correctness of a processor is often only given as long as a certain critical temperature is not exceeded. Therefore, thermal-aware mapping optimization algorithms have to include the worst-case chip temperature in their objective function. Second, as the formal thermal analysis method is several orders of magnitude faster than cycle-accurate simulation, it enables a much faster and more complete exploration of the design space.

5.8.5 Design Space Exploration

In the third case study, we apply the proposed thermal analysis methods to explore the design space of various synthetic and real-world applications.



(a) MJPEG decoder application. **(b)** Matrix multiplication application.

Fig. 5.19: Worst-case chip temperature vs. latency for 50 random mapping configurations.

Worst-Case Chip Temperature vs. Latency

We start by comparing the temporal and thermal characteristics of two real-world applications that are mapped onto a target platform with three cores. To this end, we use a prototype implementation of our mapping optimization framework to explore parts of the design space of the MJPEG decoder application and of the matrix multiplication application. The design space is formed by two analysis metrics, namely the worst-case chip temperature and the overall latency of the application, both calculated by the extended MPA framework.

Figure 5.19 shows the scatter plots for the two benchmark applications having the worst-case chip temperature on the horizontal axis and the latency on the vertical axis. Each of the 50 points refers to a different candidate mapping. It shows that no solution is optimal with respect to both latency and temperature. For the MJPEG decoder, the worst-case chip temperature of the candidate mapping leading to the lowest latency, i.e., 1.94 s, is 360.2 K and the candidate mapping leading to the lowest worst-case chip temperature of 342.2 K has a worst-case latency of 11.7 s. Similarly, for the matrix multiplication application, the worst-case chip temperature of the candidate mapping leading to the lowest latency is 354.9 K, but the lowest worst-case chip temperature is just 350.6 K. The Pareto-optimal candidates are highlighted in the figure. In particular, there are five Pareto-optimal candidates for the MJPEG decoder and six Pareto-optimal candidates for the matrix multiplication. For example, suppose that the target platform has a critical temperature of 355 K, i.e., exceeding this temperature might lead to functional errors. Then, the system designer would select the mapping with the lowest latency from all mappings that have a worst-case chip temperature smaller than 355 K.

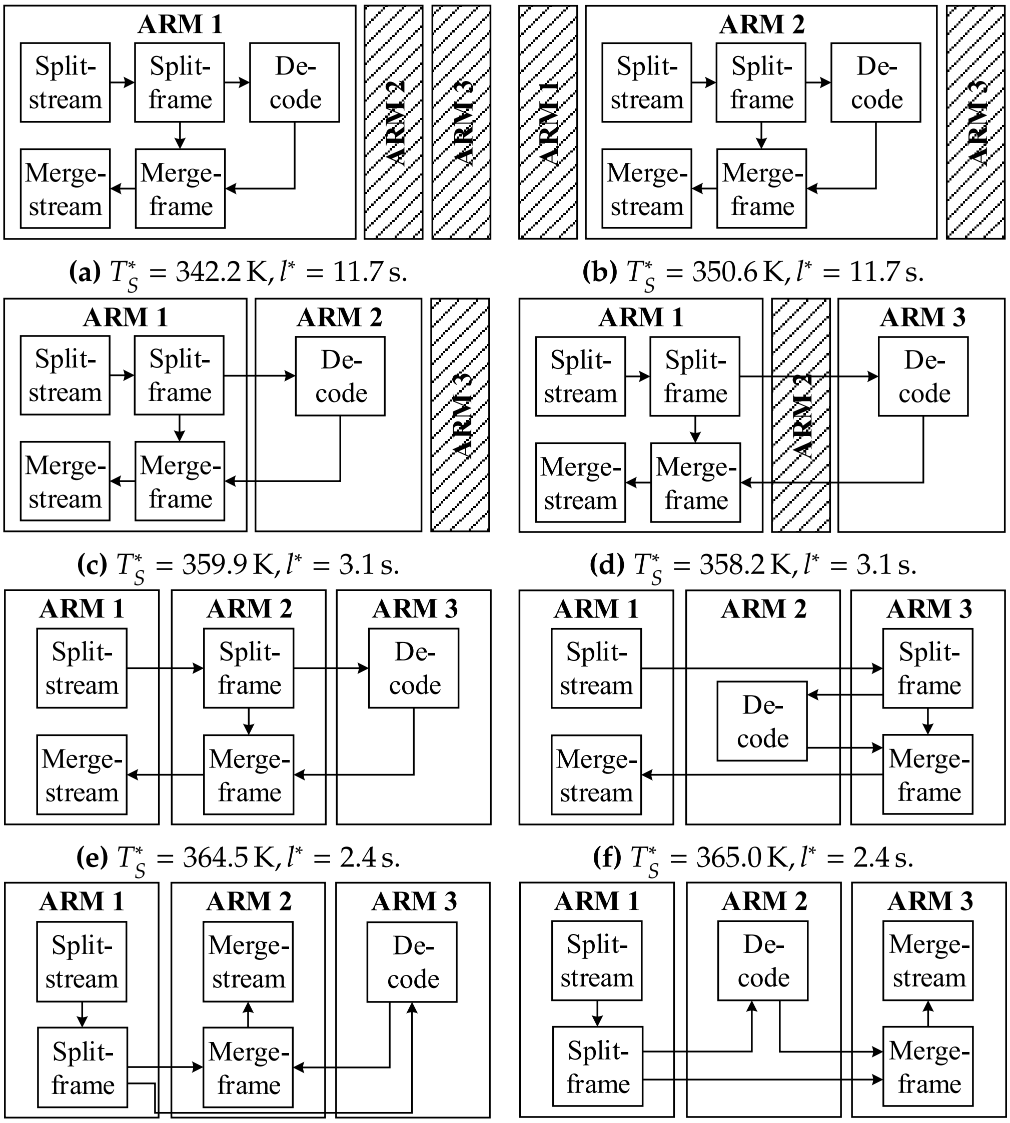
Programmed to consider this temperature constraint, the mapping optimization framework would select mapping (3) as the optimal mapping for the MJPEG decoder and mapping (6) as optimal mapping for the matrix multiplication. As the proposed algorithm offers safe bounds, the system can execute the mapping safely without further involving other dynamic thermal management strategies, which may lead to unpredictable behavior.

To study the connection between worst-case chip temperature and worst-case latency in greater detail, Fig. 5.20 outlines eight selected mapping configurations of the MJPEG decoder application together with their worst-case chip temperature T_S^* and their worst-case overall latency l^*. In order to study the effect of the placement, we study solution pairs where only the placement of the processing cores has changed. For example, mappings 5.20a and 5.20b and mappings 5.20c and 5.20d are solution pairs. It shows that the physical placement cannot be ignored in temperature analysis. In particular, mappings 5.20a and 5.20b have almost the same latency, but their peak temperatures differ by more than 8 K. Therefore, even if the mapping is already predefined, the system designer might reduce the temperature by selecting an appropriate physical placement.

Peak Temperature Reduction

Next, we apply the proposed temperature analysis methods to obtain the process-to-core assignment that minimizes the worst-case chip temperature the most. To this end, we solve the thermal optimization problem stated in Section 5.7 with different solvers and compare the obtained worst-case chip temperatures. For process networks with a small number of processes and platforms with a low number of cores, the optimization problem can be solved exhaustively. Thus, we first compare the performance of a heuristic solver with the optimal solution found by exploring the design space exhaustively. The heuristic solver uses simulated annealing [KGV83] to solve the thermal optimization problem. In addition, the average peak temperature of 20 feasible, i.e., schedulable, random process assignments is calculated.

We consider three different hardware platforms with three, four, and six cores, respectively. Each core is running at its maximum operation frequency, i.e., 1.6 GHz, and an EDF scheduler is running on each core to arbitrate between events of different processes assigned to the same core. Each process network is randomly generated so that its number of processes is between four and six. Each process v is characterized by a period, a jitter, and a computing demand. The period per_v is uniformly chosen from $[1, 400\,]ms$, the jitter is uniformly chosen from $[1\,\text{ms}, 2 \cdot per_v]$, and the computational demand is uniformly chosen from $[1, per_v \cdot f^{\max}/5]$ cycles



(a) $T_S^* = 342.2\,\mathrm{K}, l^* = 11.7\,\mathrm{s}$. **(b)** $T_S^* = 350.6\,\mathrm{K}, l^* = 11.7\,\mathrm{s}$.

(c) $T_S^* = 359.9\,\mathrm{K}, l^* = 3.1\,\mathrm{s}$. **(d)** $T_S^* = 358.2\,\mathrm{K}, l^* = 3.1\,\mathrm{s}$.

(e) $T_S^* = 364.5\,\mathrm{K}, l^* = 2.4\,\mathrm{s}$. **(f)** $T_S^* = 365.0\,\mathrm{K}, l^* = 2.4\,\mathrm{s}$.

(g) $T_S^* = 360.2\,\mathrm{K}, l^* = 2.9\,\mathrm{s}$. **(h)** $T_S^* = 360.5\,\mathrm{K}, l^* = 2.9\,\mathrm{s}$.

Fig. 5.20: Eight mapping configurations of the MJPEG decoder application together with their worst-case chip temperature T_S^* and their worst-case overall latency l^*.

with $f^{\max} = 1.6\,\mathrm{GHz}$. Finally, the real-time deadline of a process is set to its period.

Figure 5.21 compares the performance of the three solvers. Exhaustively exploring the design space results in a process-to-core assignment that has a worst-case chip temperature, which is, on average, only 0.37 K smaller than the maximum temperature of the process assignment found by simulated annealing. For comparison, the average peak temperature of the random assignments is on average 3.6 K higher than the minimum peak temperature. Calculating the optimal solution for the hardware plat-

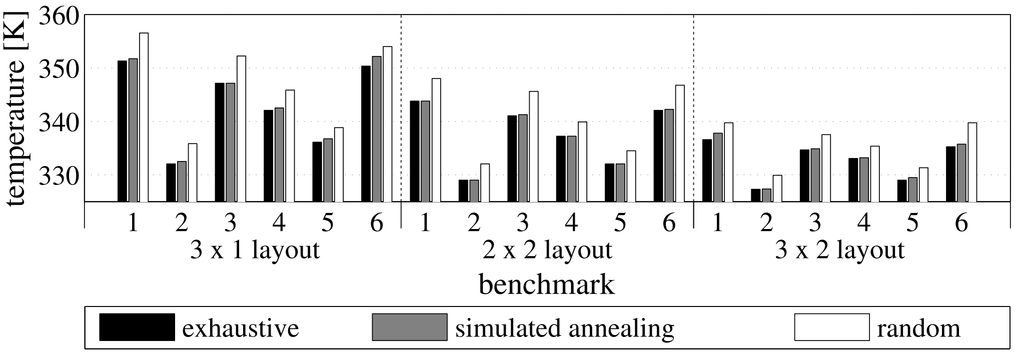


Fig. 5.21: Performance of different solvers for the temperature minimization problem. Three different hardware platforms with a 3 × 1, 2 × 2, and 3 × 2 layout are considered.

form with six cores takes on average 94.5 min and simulated annealing finishes on average in 33.8 s.

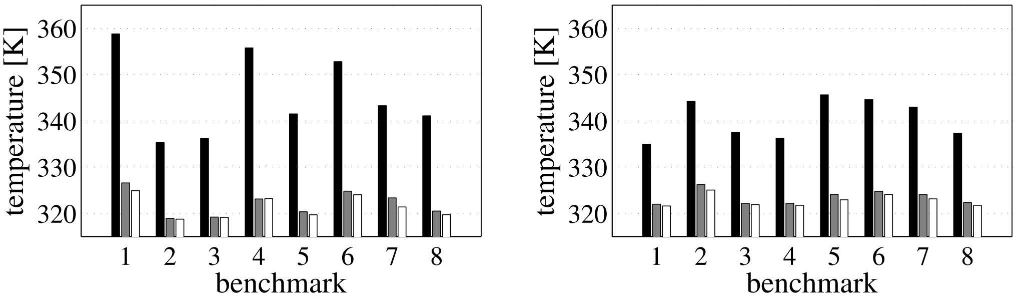
Voltage and Frequency Scaling

Finally, we evaluate the effect of frequency and voltage scaling on the worst-case chip temperature. To this end, we assume that the multi-core ARM platform has two different modes to control the operation frequency. Either all cores have a common clock domain or each core is supposed to have its own clock domain whereby the maximum operation frequency is supposed to be 1.6 GHz. Again, an EDF scheduler is running on each core to arbitrate between events of different processes assigned to the same core.

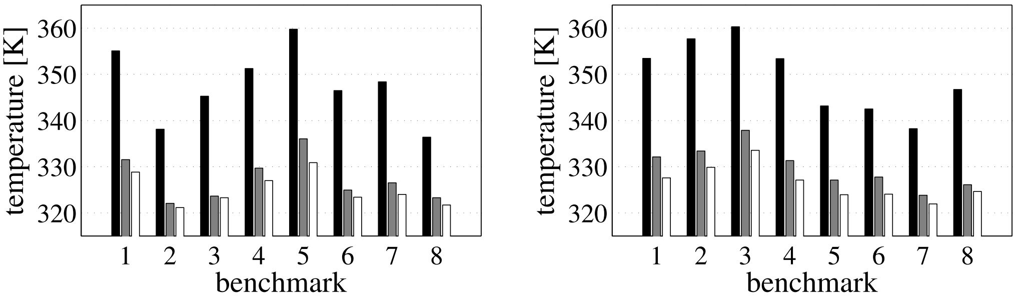
The mapping optimization framework is configured to calculate the worst-case chip temperature using Eq. (5.32) and solves the optimization problem stated in Section 5.7 for the following three configurations:

1. *Maximum frequency*: each core is running at its maximum frequency.
2. *Single clock domain*: the platform has a single clock domain for all cores and is running at the minimum operation frequency so that no real-time deadline is missed.
3. *Separate clock domain*: each core has an own clock domain and is running at the minimum operation frequency so that no real-time deadline is missed.

In other words, in the third configuration, each core has a separate frequency that is individually calculated by Eq. (5.50). In the second configuration, all cores are running at the same frequency and this frequency



(a) 3 × 1 layout. **(b)** 3 × 2 layout.



(c) 3 × 3 layout. **(d)** 4 × 4 layout.

■ max. frequency ▨ single clock domain □ separate clock domain

Fig. 5.22: Worst-case chip temperature for three different frequency configurations and four hardware platforms. The worst-case chip temperature is calculated under the following assumptions: a) all cores are running at maximum frequency, b) the platform has a single clock domain, and c) each core has a separate clock domain.

is set to the maximum frequency of all frequencies used for the third configuration.

The layouts of the considered platforms are 3 × 1, 3 × 2, 3 × 3, and 4 × 4 with 3, 6, 9, and 16 cores, respectively. We compare eight different process networks per platform and each process network is randomly generated so that it has one to three times as many processes as number of cores. We again use simulated annealing to solve the optimization problem.

In Fig. 5.22, we plot the worst-case chip temperature for the three different frequency configurations and four hardware platforms. It shows that the worst-case chip temperature can be reduced drastically when the cores are running at their optimal frequency. If each core has its own clock domain, the peak temperature is on average reduced by 24.2 K for the 3 × 1 layout, by 17.6 K for the 3 × 2 layout, 22.5 K for the 3 × 3 layout, and 22.8 K for the 4 × 4 layout.

5.9 Summary

Nowadays, the thermal wall is recognized as one of the most significant barriers towards high performance systems [HFFA11]. Reactive thermal management techniques, which are considered in general-purpose computing systems as efficient tools for temperature control, keep the maximum temperature under a given threshold by stalling or slowing down the processor. However, as they cause a significant performance degradation, reactive thermal management techniques are often undesirable in embedded systems, in particular when real-time constraints are tackled. Therefore, providing guarantees on maximum temperature is as important as functional correctness and timeliness when designing embedded many-core SoCs.

The outcome of this chapter is a high-level optimization framework for mapping dataflow process networks [LP95] onto embedded many-core SoC platforms that guarantees the final performance and correct function of the system, considering both temporal and thermal properties. In comparison with reactive thermal management techniques, the optimization framework aims at identifying a mapping alternative that results in a system that has a formally proven maximum temperature that is lower than the critical temperature of the chip. Alternatively, the mapping optimization framework aims at identifying a mapping candidate that minimizes the maximum chip temperature.

To this end, a two-stage approach is applied. First, potential mapping candidates are evaluated by means of formal worst-case analysis methods that provide safe bounds on the execution time and the maximum chip temperature. Afterwards, mapping alternatives that do not conform to real-time and peak temperature requirements are ruled out during design space exploration. In order to use formal timing and thermal analysis methods during design space exploration, the analysis models are generated automatically from the same set of specifications that are used for software synthesis. Afterwards, the analysis models are calibrated with performance data reflecting the execution of the system on the target platform. The performance data is obtained automatically prior to design space exploration based on a set of benchmark mappings. The considered thermal analysis model is able to address various thermal effects like the heat exchange between neighboring cores and temperature-dependent leakage power to model the thermal behavior of modern many-core SoC platforms accurately. Real-time calculus [TCN00], a formal method for schedulability and performance analysis of real-time systems, is applied to upper bound the workload that might arrive in any time interval. A novel thermal analysis method then identifies the critical workload trace that leads to the worst-case chip temperature, i.e., the maximum chip temperature under all feasible scenarios of task arrivals.

Finally, based on a prototype implementation of the proposed high-level mapping optimization framework, we have demonstrated that there is no single optimal solution with respect to both real-time performance and temperature, but all generated solutions are worst-case guaranteed with respect to application behavior and impact of a non-deterministic environment. With the proposed framework, system designers receive a powerful tool to map applications onto many-core SoC platforms so that the system can safely execute without involving further dynamic thermal management strategies, which may lead to unpredictable behavior.

6

Conclusion

In this chapter, we summarize the contributions of this thesis and discuss potential directions for future research.

6.1 Main Results

The aim of this thesis is to show that the properties of streaming programming models can be leveraged to develop a design, optimization, and synthesis process for embedded many-core Systems-on-Chip (SoCs) that enables the system to utilize its computing power efficiently. To this end, the Distributed Application Layer (DAL) design flow, a model-driven development approach to design reliable and efficient many-core SoCs, is proposed. The DAL design flow addresses new challenges stemming from the massive computational demand of future embedded applications and the massive hardware parallelism of many-core SoC platforms. The main contributions are summarized in the following:

- We introduce a scenario-based design flow for mapping a set of dynamically interacting streaming applications onto a heterogeneous many-core SoC. We adapt a finite state machine to specify the interactions between the applications. Each state of the finite state machine represents an execution scenario, i.e., a certain use case of the system with a predefined set of running applications. To exploit the available hardware resources efficiently, we propose a hybrid design time and runtime mapping strategy. As each execution scenario can be analyzed separately, reasoning about correctness and performance is enabled at design time. To include the evaluation of all possible failure scenarios in the design time analysis, spare architectural units are allocated during design time optimization and are used as target for process migration after the occurrence of a fault. We develop a prototype implementation of DAL targeting Intel's Single-chip Cloud Computer (SCC) processor as a proof-of-concept of the proposed design flow and runtime environment.

- We argue that process networks can be specified in a manner that enables the automatic exploration of task, data, and pipeline parallelism. To this end, we propose the semantics of Expandable Process Networks (EPNs), which extends conventional programming models for streaming applications in the sense that several possible granularities are abstracted in a single high-level specification. We show that the EPN semantics facilitates the synthesis of multiple design implementations that are all derived from the same application specification. To include the proposed concepts in the system design, we extend the DAL design flow by an additional design step, which optimizes the application structure to match the available hardware parallelism.

- We propose a systematic approach to exploit the multi-level parallelism of heterogeneous systems. The basic idea of the approach is to process multiple firings of a process concurrently and to calculate independent output tokens in parallel so that Single Instruction, Multiple Data (SIMD) execution can be achieved. As a proof-of-concept, we developed a general code synthesis framework to execute DAL applications on any Open Computing Language (OpenCL)-capable platform. In contrast to previous work, our framework is capable of embedding the parallel entities of streaming applications into low-level OpenCL kernels automatically.

- We describe a high-level optimization framework for mapping process networks onto embedded many-core architectures that optimizes a system design with respect to both performance and temperature. To evaluate the thermal characteristics of mapping candidates, we propose a novel technique to calculate a non-trivial upper bound on the peak temperature of a many-core SoC. The method is based on real-time calculus and identifies the critical workload trace that leads to the worst-case chip temperature. To integrate this formal worst-case real-time analysis method seamlessly into system design, we describe an approach to generate the underlying analysis models automatically from the same set of specifications as used for software synthesis.

6.2 Possible Future Directions

The contributions of this thesis integrate multi-application support and thermal analysis successfully into the design flow of embedded many-core SoCs. Nevertheless, there exists potential for further extensions and improvements.

The following list suggests research in several directions that bears potential for future work.

- **Handling Previously Unknown System Dynamics**

  Nowadays, a widely accepted assumption in the embedded system community is that all possible use-cases of the system are known at design time. However, this assumption is only true as long as the system has no external communication capabilities, which allow users to modify the original system configuration, e.g., by installing additional applications. If unknown applications can be installed after deployment, system analysis and mapping decisions must be performed at runtime. However, applying the previously developed mapping and optimization strategies at runtime is a challenging task.

  In the context of this thesis, three questions immediately arise. First, how to integrate and analyze new execution scenarios at runtime without causing long response times? Second, how to adapt the application's degree of parallelism at runtime, i.e., how to determine an alternative process network and its mapping to the available resources at runtime without interrupting the execution of the system? Finally, how to control the on-chip temperature and reduce the energy consumption of an embedded system efficiently if the workload changes dynamically? A promising approach for answering the third question might be to predict the thermal dynamics in real time and to trigger appropriate thermal management techniques as proposed in [YSW13].

- **Offloading Data Processing**

  Another advantage of the external communication capabilities is the possibility to offload computational intensive tasks from mobile devices to the cloud [TC13]. However, deciding which tasks should be offloaded is a challenging problem, in particular if real-time constraints must be met. Besides offloading data processing tasks, one can also think of offloading management and control tasks like runtime mapping optimization.

- **Heterogeneous Mapping Strategies**

  In Chapter 4, we have shown a systematic approach to exploit the multi-level hardware parallelism of heterogeneous systems. However, we have also observed that the performance depends on the selected mapping of application elements onto computation and communication resources and on the low-level configuration of the individual compute devices. Clearly, finding a good mapping for such a platform is considerably more difficult than finding a good mapping for a homogeneous platform. Therefore, it would be highly beneficial to have analysis models that incorporate the multi-level hardware parallelism offered by future heterogeneous systems.

- **Efficient Thermal Calibration Methods**

  In Chapter 5, we have described a method to calibrate the considered thermal analysis models automatically. A drawback of the proposed calibration method is that it requires detailed knowledge of the floorplan, the electrical characteristics, and the power consumption of the processors, which are typically not available for a real hardware platform. Therefore, it would be useful to extend the proposed calibration method to estimate the thermal characteristics of the chip accurately, without knowing the previously mentioned low-level information. A promising first approach towards this direction has been proposed in [RYBT12], where the thermal characteristics of a SoC are estimated based on a set of application-specific calibration runs. However, it is an open question if the obtained results can be used to estimate the worst-case chip temperature accurately.

- **Entering the Dark Silicon Era**

  It is likely that the area in future power-constrained processors cannot be fully utilized anymore. A phenomenon that is referred to as "dark silicon" [HFFA11, EBSA$^+$11]. Various approaches to utilize the available hardware efficiently have been proposed in literature, including implementing specialized cores [Tay12] or reconfigurable hardware [GHSV$^+$11]. How to integrate the concept of dark silicon into the system design process, however, is still an open question.

List of Acronyms

API	Application-Programming Interface.
APU	Accelerated Processing Unit.
ASIP	Application-Specific Instruction-Set processor.
AVX	Advanced Vector Extensions.
CPN	C for Process Networks.
CPU	Central Processing Unit.
CSDF	Cycle-Static Dataflow.
CU	Compute Unit.
CUDA	Compute Unified Device Architecture.
DAL	Distributed Application Layer.
DM	Deadline-Monotonic.
DMA	Direct Memory Access.
DOL	Distributed Operation Layer.
dOpenCL	Distributed OpenCL.
DSP	Digital Signal Processor.
DVFS	Dynamic Voltage and Frequency Scaling.
EA	Evolutionary Algorithm.
EDF	Earliest-Deadline-First.
EPN	Expandable Process Network.
FCFS	First-Come-First-Served.
FFT	Fast Fourier Transform.
FIFO	First-In First-Out.
FP	Fixed-Priority.
FPGA	Field Programmable Gate Array.
GPC	Greedy Processing Component.
GPU	Graphics Processing Unit.

HAM	Host Accessible Memory.
KPN	Kahn Process Network.
LTI	Linear and Time-Invariant.
MJPEG	Motion JPEG.
MPA	Modular Performance Analysis.
MPARM	Multiprocessor ARM.
MPB	Message Passing Buffer.
MPI	Message Passing Interface.
MPSoC	Multiprocessor System-on-Chip.
NoC	Network on Chip.
OpenCL	Open Computing Language.
PE	Processing Element.
PiP	Picture-in-Picture.
POSIX	Portable Operating System Interface.
RISC	Reduced Instruction Set Computer.
RM	Rate-Monotonic.
RPN	Reactive Process Network.
SCC	Single-chip Cloud Computer.
SDF	Synchronous Dataflow.
SIMD	Single Instruction, Multiple Data.
SoC	System-on-Chip.
SPDF	Synchronous Piggybacked Dataflow.
TDMA	Time Division Multiple Access.
UML	Unified Modeling Language.

Bibliography

[ABCR10] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Proc. Int'l Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 89–108, Reno, Nevada, USA, October 2010. ACM.

[ABF+13] Roberto Ammendola, Andrea Biagionil, Ottorino Frezza, Francesca Lo Cicero, Alessandro Lonardo, Pier Stanislao Paolucci, Davide Rossetti, Francesco Simula, Laura Tosoratto, and Piero Vicini. Design and Implementation of a Modular, Low Latency, Fault-Aware, FPGA-based Network Interface. In *Proc. Int'l Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–6, Cancun, Mexico, December 2013. IEEE.

[ABRW91] Neil C. Audsley, Alan Burns, Mike F. Richardson, and Andy J. Wellings. Real-Time Scheduling: The Deadline-Monotonic Approach. In *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, Atlanta, Georgia, USA, May 1991. IEEE.

[ADVP+07] David Atienza, Pablo G. Del Valle, Giacomo Paci, Francesco Poletti, Luca Benini, Giovanni De Micheli, Jose M. Mendias, and Roman Hermida. HW-SW Emulation Framework for Temperature-Aware Design in MPSoCs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):26:1–26:26, May 2007.

[AMD13] AMD. *AMD Embedded G-Series APU Platform*, 2013.

[AOB93] Bülent Abali, Fusun Ozguner, and Abdulla Bataineh. Balanced Parallel Sort on Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):572–581, May 1993.

[Bam14] Mohamed Ahmed Bamakhrama. *On Hard Real-Time Scheduling of Cyclo-Static Dataflow and its Application in System-Level Design*. PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), Faculty of Science, Leiden University, March 2014.

[BBB+05] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Journal of VLSI Signal Processing Systems*, 41(2):169–182, September 2005.

[BBM08] Luca Benini, Davide Bertozzi, and Michela Milano. Resource Management Policy Handling Multiple Use-Cases in MPSoC Platforms Using Constraint Programming. In *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 470–484. Springer, 2008.

[BCT+10] Andrea Bartolini, Matteo Cacciari, Andrea Tilli, Luca Benini, and Matthias Gries. A Virtual Platform Environment for Exploring Power, Thermal and Reliability Management Control Strategies in High-Performance Multicores. In *Proc. Great Lakes Symposium on VLSI (GLSVLSI)*, pages 311–316, Providence, Rhode Island, USA, 2010. ACM.

[BFFM12] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an Ecosystem for a Scalable, Modular and High-Efficiency Embedded Computing Accelerator. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 983–987, Dresden, Germany, March 2012. IEEE.

[BH01] Twan Basten and Jan Hoogerbrugge. Efficient Execution of Process Networks. In *Proc. Communicating Process Architectures (CPA)*, pages 1–14, Bristol, United Kingdom, September 2001. IOS Press.

[BHHT10] Iuliana Bacivarov, Wolfgang Haid, Kai Huang, and Lothar Thiele. Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-On-Chip. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 1007–1040. Springer, October 2010.

[BJ08] W. Lloyd Bircher and Lizy K. John. Analysis of Dynamic Power Management on Multi-Core Processors. In *Proc.*

Int'l Conference on Supercomputing (ICS), pages 327–338, Island of Kos, Greece, June 2008. ACM.

[BK11a] Ana Balevic and Bart Kienhuis. An Efficient Stream Buffer Mechanism for Dataflow Execution on Heterogeneous Platforms with GPUs. In *Proc. Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, pages 53–57, Galveston Island, Texas, USA, October 2011. IEEE.

[BK11b] Ana Balevic and Bart Kienhuis. KPN2GPU: An Approach for Discovery and Exploitation of Fine-grain Data Parallelism in Process Networks. *ACM SIGARCH Computer Architecture News*, 39(4):66–71, September 2011.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[BL13] Dai Bui and Edward A. Lee. StreaMorph: A Case for Synthesizing Energy-Efficient Adaptive Programs Using High-Level Abstractions. In *Proc. Int'l Conference on Embedded Software (EMSOFT)*, pages 20:1–20:10, Montreal, Quebec, Canada, October 2013. IEEE.

[BLL+05] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Yang Zhao, Haiyang Zheng, Shuvra S. Bhattacharyya, Elaine Cheong, Mudit Goel, Bart Kienhuis, Jie Liu, et al. Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California at Berkeley, 2005.

[BLTZ03] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. PISA – A Platform and Programming Language Independent Interface for Search Algorithms. In *Evolutionary Multi-Criterion Optimization*, volume 2632 of *Lecture Notes in Computer Science*, pages 494–508. Springer, 2003.

[BM01] David Brooks and Margaret Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *Proc. Int'l Symposium on High-Performance Computer Architecture (HPCA)*, pages 171–182, Monterrey, Mexico, 2001. IEEE.

[BMR90] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *Proc. Real-Time Systems Symposium*

(RTSS), pages 182–190, Lake Buena Vista, Florida, USA, December 1990. IEEE.

[Bor07] Shekhar Borkar. Thousand Core Chips: A Technology Perspective. In *Proc. Design Automation Conference (DAC)*, pages 746–749, San Diego, California, USA, June 2007. ACM.

[Bou14] Jani Boutellier. User Guide for Orcc DAL Backend. Available: `https://github.com/orcc/orcc/wiki/User-guide-for-Orcc-DAL-backend`, January 2014.

[BSP06] James Bigler, Abe Stephens, and Steven G. Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proc. Symposium on Interactive Ray Tracing*, pages 187–196, Salt Lake City, Utah, USA, September 2006. IEEE.

[BV58] Garrett Birkhoff and Richard S. Varga. Reactor Criticality and Nonnegative Matrices. *Journal of the Society for Industrial & Applied Mathematics*, 6(4):354–377, 1958.

[BZNS12] Mohamed A. Bamakhrama, Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov. A Methodology for Automated Design of Hard-real-time Embedded Streaming Systems. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 941–946, Dresden, Germany, March 2012. IEEE.

[Cas13] Jeronimo Castrillon. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. PhD thesis, RWTH Aachen Univeristy, Chair for Software for Systems on Silicon, April 2013.

[CCS+08] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. MAPS: An Integrated Framework for MPSoC Application Parallelization. In *Proc. Design Automation Conference (DAC)*, pages 754–759, Anaheim, California, USA, June 2008. ACM.

[CDH08] Thidapat Chantem, Robert P. Dick, and Xiaobo Sharon Hu. Temperature-Aware Scheduling and Assignment for Hard Real-Time Applications on MPSoCs. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 288–293, Munich, Germany, March 2008. IEEE.

[CFGK13] Marcello Coppola, Babak Falsafi, John Goodacre, and George Kornaros. From Embedded Multi-core SoCs to Scale-out Processors. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 947–951, Grenoble, France, March 2013. IEEE.

[CJS+02] Junwei Cao, Stephen A. Jarvis, Subhash Saini, Darren J. Kerbyson, and Graham R. Nudd. ARMS: An Agent-Based Resource Management System for Grid Computing. *Scientific Programming*, 10(2):135–148, 2002.

[CJVDP07] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[CLA13] Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, February 2013.

[CLB+12] Samarjit Chakraborty, Martin Lukasiewycz, Christian Buckl, Suhaib Fahmy, Naehyuck Chang, Sangyoung Park, Younghyun Kim, Patrick Leteinturier, and Hans Adlkofer. Embedded Systems and Software Challenges in Electric Vehicles. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 424–429, Dresden, Germany, March 2012. IEEE.

[CLR+01] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[CLRB11] Carsten Clauss, Stefan Lankes, Pablo Reble, and Thomas Bemmerl. Evaluation and Improvements of Programming Models for the Intel SCC Many-Core Processor. In *Proc. Int'l Conference on High Performance Computing and Simulation (HPCS)*, pages 525–532, Istanbul, Turkey, July 2011. IEEE.

[CLS+12] Yoonseo Choi, Cheng-Hong Li, Dilma Da Silva, Alan Bivens, and Eugen Schenfeld. Adaptive Task Duplication using On-Line Bottleneck Detection for Streaming Applications. In *Proc. Conference on Computing Frontiers (CF)*, pages 163–172, Cagliari, Italy, May 2012. ACM.

[CPMB14] Francesco Conti, Chuck Pilkington, Andrea Marongiu, and Luca Benini. He-P2012: Architectural Heterogeneity Exploration on a Scalable Many-core Platform. In *Proc. Great Lakes Symposium on VLSI (GLSVLSI)*, pages 231–232, Houston, Texas, USA, May 2014. ACM.

[CRW07] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Keith Whisnant. Temperature Aware Task Scheduling in MPSoCs. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1659–1664, Nice, France, April 2007. IEEE.

[CRWG08] Ayse Kivilcim Coskun, Tajana Simunic Rosing, Keith A. Whisnant, and Kenny C. Gross. Static and Dynamic Temperature-Aware Scheduling for Multiprocessor SoCs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(9):1127–1140, 2008.

[CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[CVS+10] Jeronimo Castrillon, Ricardo Velasquez, Anastasia Stulova, Weihua Sheng, Jianjiang Ceng, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Trace-Based KPN Composability Analysis for Mapping Simultaneous Applications to MPSoC Platforms. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 753–758, Dresden, Germany, March 2010. IEEE.

[DCR+12] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a High-Level Language for GPUs: (via Language Support for Architectures and Compilers). In *Proc. Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Beijing, China, June 2012. ACM/IEEE.

[DGL+13] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. In *Proc. Int'l Conference on Computational Science (ICCS)*, pages 1654–1663, Barcelona, Spain, June 2013.

[DM06] James Donald and Margaret Martonosi. Techniques for Multicore Thermal Management: Classification and New

Exploration. In *Proc. Int'l Symposium on Computer Architecture (ISCA)*, pages 78–88, Boston, Massachusetts, USA, June 2006.

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proc. Int'l Symposium on Computer Architecture (ISCA)*, pages 365–376, San Jose, California, USA, June 2011. IEEE.

[ET06] Stephan A. Edwards and Olivier Tardieu. SHIM: A Determinstic Model for Heterogeneous Embedded Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, August 2006.

[FCWT09] Nathan Fisher, Jian-Jia Chen, Shengquan Wang, and Lothar Thiele. Thermal-Aware Global Real-Time Scheduling on Multicore Systems. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 131–140, San Francisco, California, USA, April 2009. IEEE.

[Fly72] Michael Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[Fri86] Bernard Friedland. *Control Systems Design: An Introduction to State-Space Methods*. McGraw-Hill, 1986.

[GAC+13] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, Andrew Nelson, and Shubhendu Sinha. Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow. *ACM SIGBED Review*, 10(3):23–34, October 2013.

[GB03] Marc Geilen and Twan Basten. Requirements on the Execution of Kahn Process Networks. In *Programming Languages and Systems*, volume 2618 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2003.

[GB04] Marc Geilen and Twan Basten. Reactive Process Networks. In *Proc. Int'l Conference on Embedded Software (EMSOFT)*, pages 137–146, Pisa, Italy, September 2004. ACM.

[GBC06] Stefan Valentin Gheorghita, Twan Basten, and Henk Corporaal. Application Scenarios in Streaming-Oriented Embedded System Design. In *Proc. Int'l Symposium on System-on-Chip (SoC)*, pages 1–4, Tampere, Finland, November 2006.

[GCC⁺14] Leonardo Bautista Gomez, Franck Cappello, Luigi Carro, Nathan A. DeBardeleben, Bo Fang, Sudhanva Gurumurthi, Karthik Pattabiraman, Paolo Rech, and Matteo Sonza Reorda. GPGPUs: How to Combine High Computational Power with High Reliability. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1–9, Dresden, Germany, March 2014. IEEE.

[GdVA10] Pablo Garcia del Valle and David Atienza. Emulation-Based Transient Thermal Modeling of 2D/3D Systems-on-Chip with Active Cooling. *Microelectronics Journal*, 41(10):1–9, 2010.

[GHSV⁺11] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Bedford Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro*, 31(2):86–95, 2011.

[GPH⁺09] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-Scenario-Based Design of Dynamic Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):3:1–3:45, 2009.

[GTA06] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proc. Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, San Jose, California, USA, October 2006.

[GTP97] Rohini Gupta, Bogdan Tutuianu, and Lawrence T. Pileggi. The Elmore Delay as a Bound for RC Trees with Generalized Input Signals. *IEEE Transactions on Computer-Aided*

Design of Integrated Circuits and Systems, 16(1):95–104, January 1997.

[Hai10] Wolfgang Haid. *Design and Performance Analysis of Multiprocessor Streaming Applications*. PhD thesis, ETH Zurich, October 2010.

[HCK$^+$09] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. In *Proc. Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 214–223, Raleigh, North Carolina, USA, September 2009. IEEE.

[HDH$^+$10] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proc. Int'l. Solid-State Circuits Conference (ISSCC)*, pages 108–109, San Francisco, California, USA, February 2010.

[HFFA11] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6–15, 2011.

[HGBH09] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A Template for Composable and Predictable Multi-processor System on Chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2:1–2:24, 2009.

[HGT07] Kai Huang, David Grunert, and Lothar Thiele. Windowed FIFOs for FPGA-based Multiprocessor Systems. In *Proc. Int'l Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 36–41, Montreal, Quebec, Canada, July 2007. IEEE.

[HGV$^+$06] Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R. Stan. HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, 2006.

[HHB$^+$12] Kai Huang, Wolfgang Haid, Iuliana Bacivarov, Matthias Keller, and Lothar Thiele. Embedding Formal Performance

Analysis into the Design Cycle of MPSoCs for Real-time Streaming Applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(1):8:1–8:23, March 2012.

[HHBT09] Kai Huang, Wolfgang Haid, Iuliana Bacivarov, and Lothar Thiele. Coupling MPARM with DOL. TIK Report 314, ETH Zürich, September 2009.

[HHJ⁺05] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System Level Performance Analysis - The SymTA/S Approach. *IEE Proceedings - Computers and Digital Techniques*, 152(2):148–166, 2005.

[HKL⁺07] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):24:1–24:25, August 2007.

[Hoa85] Charles Antony Richard Hoare. *Communicating Sequential Processes*, volume 178. Prentice-Hall Englewood Cliffs, 1985.

[HSH⁺09] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs. In *Proc. Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 35–44, Grenoble, France, October 2009. IEEE.

[HSW⁺11] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: Portable Stream Programming on Graphics Engines. In *Proc. Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–392, Newport Beach, California, USA, March 2011. ACM.

[IBC⁺06] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proc. Int'l Symposium on Microarchitecture (MICRO)*, pages 347–358, Orlando, Florida, USA, December 2006. IEEE Computer Society.

[Int14] Intel Corporation. *Intel Xeon Phi Coprocessor: Datasheet*, 3rd edition, April 2014.

[JLK+14] Hanwoong Jung, Chanhee Lee, Shin-Haeng Kang, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. Dynamic Behavior Specification and Dynamic Mapping for Real-Time Embedded Systems: HOPES Approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):135:1–135:26, April 2014.

[JYH12] Hanwoong Jung, Youngmin Yi, and Soonhoi Ha. Automatic CUDA Code Synthesis Framework for Multicore CPU and GPU Architectures. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, volume 7203 of *Lecture Notes in Computer Science*, pages 579–588. Springer, 2012.

[Kah74] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of the IFIP Congress*, volume 74, pages 471–475, 1974.

[KAL14] KALRAY. MPPA MANYCORE. Flyer, KALRAY, February 2014. `http://www.kalray.eu/IMG/pdf/FLYER_MPPA_MANYCORE.pdf`.

[KBC09] Branislav Kisačanin, Shuvra S. Bhattacharyya, and Sek Chai, editors. *Embedded Computer Vision*. Springer, 2009.

[KBL+11] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: Distributed Resource Management for On-Chip Many-Core Systems. In *Proc. Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128, Taipei, Taiwan, October 2011. ACM.

[KDVvdW97] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In *Proc. Int'l Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 338–349, Zurich, Switzerland, July 1997. IEEE Computer Society.

[KGV83] Scott Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[KHET07] Simon Künzli, Arne Hamann, Rolf Ernst, and Lothar Thiele. Combined Approach to System Level Performance Analysis of Embedded Systems. In *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 63–68, Salzburg, Austria, Oct 2007. ACM.

[Khr10] Khronos Group. *The OpenCL Specification*, 2010.

[KKO+06] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. UML-Based Multiprocessor SoC Design Framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 5:281–320, May 2006.

[KNRSV00] Kurt Keutzer, A. Richard Newton, Jan M. Rabaey, and Alberto Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[KRD00] Bart Kienhuis, Edwin Rijpkema, and Ed Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. Int'l Workshop on Hardware/Software Codesign (CODES)*, pages 13–17, San Diego, California, USA, May 2000. ACM.

[Kre99] Frank Kreith. *CRC Handbook of Thermal Engineering*. CRC Press, 1999.

[KRJ13] Junsung Kim, Ragunathan (Raj) Rajkumar, and Markus Jochim. Towards Dependable Autonomous Driving Vehicles: A System-level Approach. *ACM SIGBED Review*, 10(1):29–32, February 2013.

[KSG12] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch. dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In *Proc. Int'l Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 174–186, Shanghai, China, May 2012. IEEE.

[KSPJ06] Amit Kumar, Li Shang, Li-Shiuan Peh, and Niraj K. Jha. HybDTM: A Coordinated Hardware-Software Approach for Dynamic Thermal Management. In *Proc. Design Automation Conference (DAC)*, pages 548–553, San Francisco, California, USA, July 2006. ACM.

[KT10] Joachim Keinert and Jürgen Teich. *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*. Springer Science & Business Media, 2010.

[LBT01] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes In Computer Science*. Springer, Berlin, Heidelberg, 2001.

[LCC12] Haeseung Lee, Weijia Che, and Karam Chatha. Dynamic Scheduling of Stream Programs on Embedded Multi-core Processors. In *Proc. Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 93–102, Tampere, Finland, October 2012. ACM.

[LDSY07] Yongpan Liu, Robert P. Dick, Li Shang, and Huazhong Yang. Accurate Temperature-Dependent Integrated Circuit Leakage Power Estimation is Easy. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1526–1531, Nice, France, April 2007. IEEE.

[LHCZ13] Shigang Li, Jingyuan Hu, Xin Cheng, and Chongchong Zhao. Asynchronous Work Stealing on Distributed Memory Systems. In *Proc. Euromicro Int'l Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 198–202, Belfast, Northern Ireland, February 2013. IEEE.

[LKP$^+$10] Chanhee Lee, Hokeun Kim, Hae-woo Park, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. A Task Remapping Technique for Reliable Multi-Core Embedded Systems. In *Proc. Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 307–316, Scottsdale, Arizona, USA, October 2010. ACM.

[LL73] Chung Laung Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, January 1973.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[Loh08] Gabriel H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proc. Int'l Symposium on Computer*

Architecture (ISCA), pages 453–464, Beijing, China, June 2008. IEEE.

[LP95] Edward A. Lee and Thomas M Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[LPCZN13] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and Efficient Work-stealing for Weak Memory Models. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 69–80, Shenzhen, China, February 2013. ACM.

[MAR10] Marco Mattavelli, Ihab Amer, and Mickaël Raulet. The Reconfigurable Video Coding Standard [Standards in a Nutshell]. *IEEE Signal Processing Magazine*, 27(3):159–167, May 2010.

[MAV+10] Giovanni Mariani, Prabhat Avasare, Geert Vanmeerbeeck, Chantal Ykman-Couvreur, Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. An Industrial Design Space Exploration Framework for Supporting Run-Time Resource Management on Multi-Core Systems. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 196–201, Dresden, Germany, March 2010. IEEE.

[MBF+12] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a Many-Core Computing Accelerator for Embedded SoCs: Performance Evaluation of Visual Analytics Applications. In *Proc. Design Automation Conference (DAC)*, pages 1137–1142, San Francisco, California, USA, June 2012. ACM.

[MBM97] Marco Mattavelli, Sylvain Brunetton, and Daniel Mlynek. A Parallel Multimedia Processor for Macroblock Based Compression Standards. In *Proc. Int'l Conference on Image Processing*, volume 2, pages 570–573, Santa Barbara, California, USA, October 1997. IEEE.

[MC14] Orlando Moreira and Henk Corporaal. *Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor*, volume 24 of *Embedded Systems*. Springer, 2014.

[MCP+12] Bryan Marker, Ernie Chan, Jack Poulson, Robert Geijn, Rob F. Van der Wijngaart, Timothy G. Mattson, and

Theodore E. Kubaska. Programming Many-Core Architectures - A Case Study: Dense Matrix Computations on the Intel Single-chip Cloud Computer Processor. *Concurrency and Computation: Practice and Experience*, 24(12):1317–1333, August 2012.

[Mea55] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MGC08] John L. Manferdelli, Naga K. Govindaraju, and Chris Crall. Challenges and Opportunities in Many-Core Computing. *Proceedings of the IEEE*, 96(5):808–815, May 2008.

[MKK77] Hajime Maeda, Shinzo Kodama, and Fumihiko Kajiya. Compartmental System Analysis: Realization of a Class of Linear Systems with Physical Constraints. *IEEE Transactions on Circuits and Systems*, 24(1):8–14, 1977.

[ML94] Praveen K. Murthy and Edward A. Lee. On the Optimal Blocking Factor for Blocked, Non-Overlapped Multiprocessor Schedules. In *Proc. of Asilomar Conference on Signals, Systems and Computers (ASSC)*, volume 2, pages 1052–1057, Pacific Grove, California, USA, October 1994. IEEE.

[MMA+07] Srinivasan Murali, Almir Mutapcic, David Atienza, Rajesh Gupta, Stephen Boyd, and Giovanni De Micheli. Temperature-Aware Processor Frequency Assignment for MPSoCs Using Convex Optimization. In *Proc. Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 111–116, Salzburg, Austria, October 2007. IEEE.

[Moo56] Edward F. Moore. Gedanken-Experiments on Sequential Machines. In Claude Elwood Shannon and John McCarthy, editors, *Automata Studies: Annals of Mathematical Studies*, volume 34, pages 129–153. Princeton University Press, 1956.

[MXK+13] Takashi Miyamori, Hui Xu, Takeshi Kodaka, Hiroyuki Usui, Toru Sano, and Jun Tanabe. Development of Low Power Many-core SoC for Multimedia Applications. In *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, pages 773–777, Grenoble, France, March 2013.

[NSD08] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):542–555, 2008.

[Nvi08] CUDA Programming Guide. NVIDIA, 2008.

[NVI14] NVIDIA. NVIDIA Tegra K1 - A New Era in Mobile Computing. White paper, NVIDIA, January 2014.

[Pac96] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1996.

[PBG⁺13] Pier Stanislao Paolucci, Iuliana Bacivarov, Gert Goossens, Rainer Leupers, Frédéric Rousseau, Christoph Schumacher, Lothar Thiele, and Piero Vicini. EURETILE 2010-2012 Summary: First Three Years of Activity of the European Reference Tiled Experiment. arXiv:1305.1459 [cs.DC], 2013.

[PJH02] Chanik Park, Jaewoong Jung, and Soonhoi Ha. Extended Synchronous Dataflow for Efficient DSP System Prototyping. *Design Automation for Embedded Systems*, 6(3):295–322, 2002.

[PJL⁺06] Pier Stanislao Paolucci, Ahmed A. Jerraya, Rainer Leupers, Lothar Thiele, and Piero Vicini. SHAPES: A Tiled Scalable Software Hardware Architecture Platform for Embedded Systems. In *Proc. Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 167–172, Seoul, Korea, October 2006. ACM.

[PL95] José Luis Pino and Edward A. Lee. Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors. In *Proc. Int'l Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 4, pages 2643–2646, Detroit, Michigan, USA, May 1995. IEEE.

[PP96] Alberto Pettorossi and Maurizio Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys (CSUR)*, 28(2):360–414, June 1996.

[Qua14a] Qualcomm. *Qualcomm Snapdragon 805 Processor*, 2014. Available online: `www.qualcomm.com/media/documents/files/snapdragon-805-product-brief.pdf`.

[Qua14b] Qualcomm. Snapdragon 615 Processor Specs. `http://www.qualcomm.com/snapdragon/processors/615`, May 2014.

[Ram11] Carl Ramey. Tile-Gx-100 ManyCore Processor: Acceleration Interfaces and Architecture. Presented at HotChips 23, August 2011.

[RCN08] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits*. Prentice Hall Press, 3rd edition, 2008.

[RYB+11] Devendra Rai, Hoeseok Yang, Iuliana Bacivarov, Jian-Jia Chen, and Lothar Thiele. Worst-Case Temperature Analysis for Real-Time Systems. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1–6, Grenoble, France, March 2011. IEEE.

[RYBT12] Devendra Rai, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele. Power Agnostic Technique for Efficient Temperature Estimation of Multicore Embedded Systems. In *Proc. Int'l Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 61–70, Tampere, Finland, October 2012. ACM.

[SABC10] Yang Sun, Kiarash Amiri, Michael Brogioli, and Joseph R. Cavallaro. Application-Specific Accelerators for Communications. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 329–362. Springer, 2010.

[SBGC07] Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. Multiprocessor Resource Allocation for Throughput-constrained Synchronous Dataflow Graphs. In *Proc. Design Automation Conference (DAC)*, pages 777–782, San Diego, California, USA, June 2007. ACM.

[Sch11] Lars Schor. Thermal Simulation and Analysis Methods for Many-Core Platforms. Master's thesis, ETH Zurich, Zurich, Switzerland, January 2011.

[SCT10] Andreas Schranzhofer, Jian-Jian Chen, and Lothar Thiele. Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms. *IEEE Transactions on Industrial Informatics*, 6(4):692–707, November 2010.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, September/October 2003.

[Ser72] Omri Serlin. Scheduling of Time Critical Processes. In *Proc. of Spring Joint Computer Conference (AFIPS)*, pages 925–932, Atlantic City, New Jersey, USA, May 1972. ACM.

[SGA+13] Robert Soule, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel. Dynamic Expressivity with Static Optimization for Streaming Languages. In *Proc. Int'l Conference on Distributed Event-Based Systems (DEBS)*, pages 159–170, Arlington, Texas, USA, June 2013. ACM.

[SGB10] Sander Stuijk, Marc Geilen, and Twan Basten. A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour. In *Proc. Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, pages 548–555, Lille, France, September 2010. IEEE.

[SJ04] Ingo Sander and Axel Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 23(1):17–32, 2004.

[SKS+10] Ahsan Shabbir, Akash Kumar, Sander Stuijk, Bart Mesman, and Henk Corporaal. CA-MPSoC: An Automated Design Flow for Predictable Multi-processor Architectures for Multiple Applications. *Journal of Systems Architecture*, 56(7):265–277, July 2010.

[SKS11] Amit Kumar Singh, Akash Kumar, and Thambipillai Srikanthan. A Hybrid Strategy for Mapping Multiple Throughput-constrained Applications on MPSoCs. In *Proc. Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 175–184, Taipei, Taiwan, October 2011. ACM.

[SLA12] Anastasia Stulova, Rainer Leupers, and Gerd Ascheid. Throughput Driven Transformations of Synchronous Data Flows for Mapping to Heterogeneous MPSoCs. In *Proc. Int'l Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 144–151, Samos, Greece, July 2012. IEEE.

[SLS07] Gerald Sabin, Matthew Lang, and P. Sadayappan. Moldable Parallel Job Scheduling Using Job Efficiency: An Iterative Approach. In *Job Scheduling Strategies for Parallel Processing*, volume 4376 of *Lecture Notes in Computer Science*, pages 94–114. Springer, 2007.

[SMV10] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: Data Orchestration and Tuning for OpenCL Devices. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Proc. Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2010.

[SRV$^+$10] Mahankali Sridhar, Arvind Raj, Alessandro Vincenzi, Martino Ruggiero, Thomas Brunschwiler, and David Atienza Alonso. 3D-ICE: Fast Compact Transient Thermal Modeling For 3D-ICs with Inter-tier Liquid Cooling. In *Proc. Int'l Conference on Computer-Aided Design (ICCAD)*, pages 463–470, San Jose, California, USA, November 2010. IEEE.

[SSB$^+$12] Enqiang Sun, Dana Schaa, Richard Bagley, Norman Rubin, and David Kaeli. Enabling Task-level Scheduling on Heterogeneous Platforms. In *Proc. Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*, pages 84–93, London, United Kingdom, March 2012. ACM.

[SSS$^+$04] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-Aware Microarchitecture: Modeling and Implementation. *ACM Transactions on Architecture Code Optimization*, 1(1):94–125, 2004.

[SWNR10] Nicolas Siret, Matthieu Wipliez, Jean-Francois Nezan, and Aimad Rhatay. Hardware Code Generation from Dataflow Programs. In *Proc. Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 113–120, Edinburgh, Scotland, October 2010. IEEE.

[TA10] William Thies and Saman Amarasinghe. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *Proc. Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 365–376, Vienna, Austria, September 2010. ACM.

[Tay12] Michael B. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Proc. Design Automation Conference (DAC)*, pages 1131–1136, San Francisco, California, USA, June 2012. ACM.

[TBHH07] Lother Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proc. Int'l Conference on Application of Concurrency to System Design (ACSD)*, pages 29–40, Bratislava, Slovak Republic, July 2007. IEEE.

[TC13] Anas Toma and Jian-Jia Chen. Server Resource Reservations for Computation Offloading in Real-Time Embedded Systems. In *Proc. Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 31–39, Montreal, Quebec, Canada, October 2013. IEEE.

[TCN00] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. In *Proc. Int'l Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, Geneva, Switzerland, May 2000. IEEE.

[Thi09] William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In R. Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer, Grenoble, France, April 2002.

[URK11] Isaías A Comprés Ureña, Michael Riepen, and Michael Konow. RCKMPI – Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC). In *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2011.

[Vaj11] Andras Vajda. *Programming Many-Core Chips*. Springer, 2011.

[vdWMH11] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight Communications on Intel's Single-chip Cloud Computer Processor. *ACM SIGOPS Operating Systems Review*, 45(1):73–83, 2011.

[VHR+08] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, et al. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, January 2008.

[VNS07] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. PN: A Tool for Improved Derivation of Process Networks. *EURASIP Journal on Embedded Systems*, 2007(1):19:1–19:13, January 2007.

[Wal92] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *IEEE Transactions on Consumer Electronics*, 38(1):18–34, August 1992.

[WC93] Cheng-Wen Wu and Chen-Ti Chang. FFT Butterfly Network Design for Easy Testing. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 40(2):110–115, February 1993.

[WDW10] Yasuko Watanabe, John D. Davis, and David A. Wood. WiDGET: Wisconsin Decoupled Grid Execution Tiles. In *Proc. Int'l Symposium on Computer Architecture (ISCA)*, pages 2–13, Saint-Malo, France, June 2010. ACM.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, May 2008.

[WGHP11] Rick Weber, Akila Gothandaraman, Robert J. Hinde, and Gregory D. Peterson. Comparing Hardware Accelerators in Scientific Applications: A Case Study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, January 2011.

[WJM08] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor System-on-Chip (MPSoC) Technology.

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(10):1701–1713, October 2008.

[Wol14] Marilyn Wolf. *High-Performance Embedded Computing: Applications in Cyber-Physical Systems and Mobile Computing*. Elsevier Science, 2014.

[WTVL06] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System Architecture Evaluation Using Modular Performance Analysis: A Case Study. *Int'l Journal on Software Tools for Technology Transfer*, 8(6):649–667, November 2006.

[XH06] Yuan Xie and Wei-lun Hung. Temperature-Aware Task Allocation and Scheduling for Embedded Multiprocessor Systems-on-Chip (MPSoC) Design. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 45(3):177–189, December 2006.

[XP90] Jia Xu and David Parnas. Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, March 1990.

[XTU+12] Hui Xu, Jun Tanabe, Hiroyuki Usui, Soichiro Hosoda, Toru Sano, Kazumasa Yamamoto, Takeshi Kodaka, Nobuhiro Nonogaki, Nau Ozaki, and Takashi Miyamori. A Low Power Many-Core SoC with Two 32-Core Clusters Connected by Tree Based NoC for Multimedia Applications. In *Proc. Symposium on VLSI Circuits (VLSIC)*, pages 150–151, Honolulu, Hawaii, USA, June 2012. IEEE.

[YCTK10] Chuan-Yue Yang, Jian-Jia Chen, Lothar Thiele, and Tei-Wei Kuo. Energy-Efficient Real-Time Task Scheduling with Temperature-Dependent Leakage. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 9–14, Dresden, Germany, March 2010. IEEE.

[YH09] Hoeseok Yang and Soonhoi Ha. Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 69–74, Nice, France, April 2009. IEEE.

[YSW13] Buyoung Yun, Kang G. Shin, and Shige Wang. Predicting Thermal Behavior for Temperature Management in Time-Critical Multicore Systems. In *Proc. Real-Time and Embedded*

Technology and Applications Symposium (RTAS), pages 185–194, Philadelphia, Pennsylvania, USA, April 2013. IEEE.

[ZBS13] Jiali Teddy Zhai, Mohamed A. Bamakhrama, and Todor Stefanov. Exploiting Just-Enough Parallelism when Mapping Streaming Applications in Hard Real-Time Systems. In *Proc. Design Automation Conference (DAC)*, pages 170:1–170:8, Austin, Texas, USA, June 2013. ACM.

[ZGS+08] Changyun Zhu, Zhenyu Gu, Li Shang, Robert P. Dick, and Russ Joseph. Three-Dimensional Chip-Multiprocessor Run-Time Thermal Management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(8):1479–1492, August 2008.

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Swiss Federal Institute of Technology (ETH) Zurich, 2001.

List of Publications

The following list includes publications that form the basis of this thesis. The corresponding chapters are indicated in parentheses.

Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele. Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems. In *Proc. Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 71–80, Tampere, Finland, October 2012. ACM. (Chapter 2).

Lars Schor, Devendra Rai, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele. Reliable and Efficient Execution of Multiple Streaming Applications on Intel's SCC Processor. In Dieter Mey et al., editors, *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 790–800. Springer, 2014. (Chapter 2).

Lars Schor, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele. Expandable Process Networks to Efficiently Specify and Explore Task, Data, and Pipeline Parallelism. In *Proc. Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 5:1–5:10, Montreal, Quebec, Canada, October 2013. IEEE. (Chapter 3).

Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. AdaPNet: Adapting Process Networks in Response to Resource Variations. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 8.1:1-8.1:10, New Delhi, India, October 2014. ACM. (Chapter 3).

Lars Schor, Andreas Tretter, Tobias Scherer, and Lothar Thiele. Exploiting the Parallelism of Heterogeneous Systems using Dataflow Graphs on Top of OpenCL. In *Proc. Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 41–50, Montreal, Quebec, Canada, October 2013. IEEE. (Chapter 4).

Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. Worst-Case Temperature Guarantees for Real-Time Applications on Multi-Core Systems. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 87–96, Beijing, China, April 2012. IEEE Computer. (Chapter 5).

Lothar Thiele, Lars Schor, Iuliana Bacivarov, and Hoeseok Yang. Predictability for Timing and Temperature in Multiprocessor System-on-Chip Platforms. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(S1):48:1–48:25, March 2013. (Chapter 5).

Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. Fast Worst-Case Peak Temperature Evaluation for Real-Time Applications on Multi-Core Systems. In *Proc. Latin American Test Workshop (LATW)*, pages 1–6, Quito, Ecuador, April 2012. IEEE. (Chapter 5).

Lars Schor, Iuliana Bacivarov, Hoeseok Yang, and Lothar Thiele. Efficient Worst-Case Temperature Evaluation for Thermal-Aware Assignment of Real-Time Applications on MPSoCs. *Journal of Electronic Testing*, 29(4):521–535, 2013. (Chapter 5).

The following list includes publications that are not part of this thesis

Andres Gomez, Lars Schor, Pratyush Kumar, and Lothar Thiele. SF3P: A Framework to Explore and Prototype Hierarchical Compositions of Real-Time Schedulers. In *Proc. IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 1–7, New Delhi, India, October 2014. IEEE.

Lars Schor, Iuliana Bacivarov, Luis Gabriel Murillo, Pier Stanislao Paolucci, Frederic Rousseau, Ashraf El Antably, Robert Buecs, Nicolas Fournel, Rainer Leupers, Devendra Rai, Lothar Thiele, Laura Tosoratto, Piero Vicini, and Jan Weinstock. EURETILE Design Flow: Dynamic and Fault Tolerant Mapping of Multiple Applications onto Many-Tile Systems. In *Proc. IEEE Int'l Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 182–189, Milan, Italy, August 2014. IEEE Computer.

Devendra Rai, Lars Schor, Nikolay Stoimenov, and Lothar Thiele. Distributed Stable States for Process Networks: Algorithm, Analysis, and Experiments on Intel SCC. In *Proc. Design Automation Conference (DAC)*, pages 167:1–167:10, Austin, Texas, USA, June 2013. ACM.

Devendra Rai, Lars Schor, Nikolay Stoimenov, Iuliana Bacivarov, and Lothar Thiele. Designing Applications with Predictable Runtime Characteristics for the Baremetal Intel SCC. In Dieter an Mey et al., editors, *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 779–789. Springer, 2014.

Lars Schor, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele. Worst-Case Temperature Analysis for Different Resource Models. *IET Circuits, Devices & Systems*, 6(5):297–307, September 2012.

Lars Schor, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele. Thermal-Aware Task Assignment for Real-Time Applications on Multi-Core Systems. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 294–313. Springer, 2012.

Christian Fabre, Iuliana Bacivarov, Ananda Basu, Martino Ruggiero, David Atienza, Eric Flamand, Jean-Pierre Krimm, Julien Mottin, Lars Schor, Pratyush Kumar, Hoeseok Yang, Devesh Chokshi, Lothar Thiele, Saddek Bensalem, Marius Bozga, Mohamed Sabry, Yusuf Leblebici, Giovanni De Micheli, and Diego Melpignano. PRO3D, Programming for Future 3D Manycore Architectures: Status After 24 Months. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*. Springer, Turin, Italy, 2012.

Shin-Haeng Kang, Hoeseok Yang, Lars Schor, Iuliana Bacivarov, Soonhoi Ha, and Lothar Thiele. Multi-Objective Mapping Optimization via Problem Decomposition for Many-Core Systems. In *Proc. Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 28–37, Tampere, Finland, October 2012. IEEE.

Lars Schor, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele. Worst-Case Temperature Analysis for Different Resource Availabilities: A Case Study. In José L. Ayala, Braulio García-Cámara, Manuel Prieto, Martino Ruggiero, and Gilles Sicard, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*, volume 6951 of *Lecture Notes in Computer Science*, pages 288–297. Springer, 2011.

Lothar Thiele, Lars Schor, Hoeseok Yang, and Iuliana Bacivarov. Thermal-Aware System Analysis and Software Synthesis for Embedded Multi-Processors. In *Proc. Design Automation Conference (DAC)*, pages 268–273, San Diego, California, USA, June 2011. ACM.

Curriculum Vitae

Name	Lars Urs Schor
Date of Birth	19.10.1984
Citizen of	Subingen SO
Nationality	Swiss

Education:

2011–2014	ETH Zurich, Computer Engineering and Networks Laboratory Ph.D. studies under the supervision of Prof. Dr. Lothar Thiele Awarded with the "Intel Doctoral Student Honor Programme Award Fellowship" 2012-2013 by Intel Corporation
2008–2011	ETH Zurich Graduated as Master of Science ETH (with distinction) Awarded with the "Willi Studer Preis" 2011 by ETH Zurich Awarded with the "ETH Medal" 2011 by ETH Zurich
2005–2008	ETH Zurich Graduated as Bachelor of Science ETH (with distinction)
1999–2003	Cantonal School Solothurn Awarded with the "Bernhard Bärtschi Preis"

Professional Experience:

2011–2014	Research and Teaching Assistant at ETH Zurich
2009–2010	Software Engineer at Menlo Innovations LLC, Ann Arbor, USA
2009	Junior Research Assistant at ETH Zurich
2004–2009	Software Engineer at ITEMA (Switzerland) Ltd., Rüti
2003	Substitute Class Teacher, Bezirksschule Schützenmatt, Solothurn