Design and Performance Analysis of Multiprocessor Streaming Applications

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by
WOLFGANG CHRISTIAN FERDINAND HAID
Dipl. El.-Ing. ETH Zurich
born November 24, 1980
citizen of Austria

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Jan Madsen, co-examiner

2010
Design and Performance Analysis of Multiprocessor Streaming Applications

Wolfgang Christian Ferdinand Haid
A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

Diss. ETH No. 19276

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Jan Madsen, co-examiner

Examination date: October 4, 2010
Abstract

The processing of regularly structured data streams frequently occurs in the context of real-time audio, video, and digital signal processing in consumer electronics devices, communication systems, and medical systems, to name a few. Software programs developed for this purpose are commonly referred to as streaming applications. The steadily growing computational demand of streaming applications increasingly leads to implementations on heterogeneous single-chip multiprocessors. This trend calls for suitable design automation techniques to deal with the complexity of the resulting systems and raises the questions of how (real-time) streaming applications can be efficiently implemented on multiprocessors and how the timing behavior of these applications can be analyzed.

In this thesis, these questions are tackled by proposing the distributed operation layer (Dol) design flow that automates the implementation and performance analysis of multiprocessor streaming applications. To this end, a streaming application is specified as a dataflow process network, that is, as a network of actors that explicitly communicate over first-in first-out channels. Then, fundamental properties of dataflow process networks are leveraged to automatically transform this high-level specification into a platform-specific implementation and an analytic performance analysis model.

Specifically, the following contributions are presented in this thesis. Regarding the platform-specific implementation, a novel multiprocessor run-time environment for dataflow process networks based on so-called protothreads and windowed first-in first-out channels is introduced which considerably reduces the run-time overhead compared to existing solutions. Regarding the analytic performance analysis, a new method to analyze the timing behavior of actors with complex activation schemes is proposed which extends the modeling scope of compositional performance analysis, a class of state-of-the-art performance analysis methods for distributed real-time systems. Furthermore, approaches for the automated implementation and performance analysis of multiprocessor streaming applications are described. A prototype of the Dol design flow has been developed that is used as a proof-of-concept
to gain insights into the potential and limitations of the proposed design flow.

It is concluded that an approach based on dataflow process networks is a viable strategy for the design and performance analysis of multiprocessor streaming applications. Specifically, the automated implementation and performance analysis relieves software developers and performance analysts of the associated technical details, giving them more time to focus on essential design activities.
Zusammenfassung


In der vorliegenden Dissertation werden diese Fragen im Rahmen eines Entwurfsablaufs (distributed operation layer — DoI) behandelt, der die Implementierung von Streaming-Applikationen auf Multiprozessoren und deren Performanz-Analyse automatisiert. Dazu wird eine Streaming-Applikation als Prozessnetzwerk spezifiziert, das heisst als Netzwerk von Aktoren, die ausschliesslich über Kanäle kommunizieren. Die fundamentalen Eigenschaften von Prozessnetzwerken werden anschliessend verwendet, um diese Spezifikation automatisch in eine plattformspezifische Implementierung bzw. in ein Modell zur Performanz-Analyse zu transformieren.

Ein Prototyp des vorgeschlagenen Entwurfsablaufs wurde entwickelt, um einerseits seine Tauglichkeit zu zeigen und andererseits seine Möglichkeiten und Beschränkungen einzuschätzen.

Zusammenfassend wird festgehalten, dass ein auf Prozessnetzwerken basierter Entwurfsablauf eine praktikable Strategie darstellt, um Streaming-Applikationen auf Multiprozessoren zu implementieren und deren Performanz zu analysieren. Insbesondere entlastet die automatische Implementierung und Performanz-Analyse Software-Entwickler und Performanz-Analysten von der Aufgabe, sich mit den diesbezüglichen technischen Details befassen zu müssen, sodass sie sich auf die wesentlichen Entwurfsaufgaben konzentrieren können.
Acknowledgements

First and foremost, I would like to thank my advisor Prof. Dr. Lothar Thiele for his true encouragement and the many fruitful discussions inspiring much of the work presented in this thesis. I would also like to thank Prof. Dr. Jan Madsen for co-examining this thesis.

Many other people directly or indirectly contributed to the content of this dissertation. Especially, I would like to thank my officemate Iuliana Bacivarov and my office neighbor Kai Huang for the successful teamwork throughout the years. Central for this dissertation were also several students who I enjoyed to work with. In particular, I am very grateful to Matthias Keller and Lars Schor for their valuable contributions.

I feel very fortunate to have been part of a motivating, dedicated, and fun team at the Computer Engineering Laboratory. Dear TECies, thank you for sharing so many memorable moments during mountain days (with and without skis), during evenings at the Limmat or the Zürichsee, and at celebrations of best-paper awards, soccer victories, SOLA rankings, or simply PhD life.

Finally, my dearest thanks go to my family for their love and support, and asking me over and over again when this manuscript will eventually be finished.

The work presented in this thesis was funded in part by the EU Sixth Framework Programme project SHAPES (project number 026825). This support is gratefully acknowledged.
## Contents

Abstract i

Zusammenfassung iii

Acknowledgements v

1 Introduction 1
   1.1 Single-Chip Multiprocessors 2
   1.2 Multiprocessor Streaming Applications 3
   1.3 Model-Driven Development 5
   1.4 Contributions 8
   1.5 Thesis Outline 11

2 Distributed Operation Layer 13
   2.1 Target Architectures 14
   2.2 Efficiency, Scalability, Portability, and Predictability 16
   2.3 Programming Model 18
   2.4 DOL Software Design Flow 22
   2.5 Related Work 28
   2.6 Summary 31

3 Lightweight Multiprocessor Run-Time Environment 33
   3.1 Problem Statement 35
   3.2 Lightweight Multiprocessor Run-Time Environment 36
   3.3 Efficient Execution of Process Networks on the Cell BE 43
   3.4 Experimental Results 47
   3.5 Related Work 54
   3.6 Summary 55

4 Modeling Actors in Modular Performance Analysis 57
   4.1 Actor Model 59
   4.2 Approach 61
   4.3 Modular Performance Analysis 62
   4.4 Modeling Actors in Modular Performance Analysis 70
   4.5 Experimental Results 80
   4.6 Related Work 88
   4.7 Summary 89
5 Generation and Calibration of Performance Analysis Models 91
   5.1 Problem Statement .................................... 93
   5.2 Automated Model Generation and Calibration ........ 95
   5.3 MPA Model Generation and Calibration in DOL ....... 100
   5.4 Experimental Results ................................ 105
   5.5 Related Work ......................................... 114
   5.6 Summary .............................................. 117

6 Conclusions 119
   6.1 Summary of the Contributions ......................... 119
   6.2 Conclusions ........................................... 120
   6.3 Future Directions ...................................... 122

A Snapshots of the Design Flow of a Streaming Application 125
   A.1 System Specification .................................. 125
   A.2 Software Synthesis ................................... 130
   A.3 Automated Model Generation and Calibration ....... 132

Bibliography 135

Curriculum Vitæ 151
An embedded system can be defined as a special-purpose computer system that is closely integrated into a system that is often neither used nor perceived of as a computer. In many embedded systems, the processing of data streams constitutes a large share of the system’s overall functionality. One can observe that this functionality is increasingly implemented in digital circuits and software, rather than analog electronic circuitry. Many recent examples evidence this digitalization of embedded systems: in consumer electronics, digital video broadcasting has replaced analog terrestrial television broadcasting. In communication systems, software-defined radio has become a viable alternative for realizing multi-standard transceivers. In medical systems, ultrasound devices can be packaged into phone-sized cases due to the replacement of analog circuitry by digital (array) signal processing.

When the processing of data streams takes place in software programs, these programs are commonly referred to as streaming applications. Obviously, the complexity and performance requirements of streaming applications are continuously increasing because of the steady demand for higher quality and new features. Primarily, this leads to an increased computational demand that cannot be satisfied by unprocessors in the long term due to frequency scaling limits of CMOS chips. The focus is thus moving towards (heterogeneous) multiprocessors that are expected to scale to new CMOS technologies by integrating multiple processors on a single chip.

How to efficiently tap the performance potential of multiprocessors, however, is an open question. Commonly accepted programming models, software libraries, and design automation tools are not yet available, hence software engineers need to familiarize themselves with minute
Chapter 1. Introduction

hardware details when authoring multiprocessor applications. Similar observations apply to the performance analysis of multiprocessors, which is demanding due to the complex interaction of processors, memories, interconnects, and peripherals, and the difficulty of gathering all the data for creating faithful analysis models.

Consequently, both the design and the performance analysis of multiprocessor (streaming) applications call for automation techniques to improve programmability, performance, and predictability by relieving application developers from dealing with low-level aspects. The goal is to reduce so-called accidental (technologically imposed) complexity [Bro87] and thereby empower software developers to focus on the essential (unavoidable, problem inherent) complexity. Suitable techniques for facing this challenge are explored in this thesis.

1.1 Single-Chip Multiprocessors

Traditionally, monolithic processors of increasing complexity and clock frequencies dominated the evolution of microprocessor design. According to the International Technology Roadmap for Semiconductors (ITRS) [ITR], the trend of increasing complexity will continue in the foreseeable future. The reason is that Moore’s law still holds, stating that the number of transistors per chip doubles approximately every two years. On the other hand, microprocessor designers face technological constraints of modern deep sub-micrometer CMOS technologies that are commonly referred to as “walls” [ABC06]:

**Frequency (Power) Wall.** The power dissipation of chips increases with the clock frequency. The method of compensating for this increase by lowering the minimum feature size and reducing voltage is running out of steam and further frequency scaling yields small, if any, returns.

**Memory Wall.** With increasing clock frequencies, the latency to external memory in terms of clock cycles is increasing. This effectively creates a memory bandwidth bottleneck.

**ILP Wall.** Many architectures and corresponding compilers attempt to exploit the instruction-level parallelism (ILP) of sequential applications to increase performance. Since ILP in sequential application is limited, the performance gains of exploiting ILP are diminishing.

Unfortunately, these walls prohibit a further evolution along the traditional roadmap by scaling processor frequencies. To still be able
to sustain the raw performance increase that can be expected due to Moore’s law, microprocessor design is moving to multiprocessors. The main advantage of integrating multiple processors (cores) on a single chip is that performance increase comes from increasing the number of processors rather than frequency, thereby circumventing the three walls: even when clock frequencies only grow moderately (or not at all), the raw performance increases due to the availability of multiple processors. Therefore, in recent years all major chip manufacturers and intellectual property (IP) suppliers have started offering (heterogeneous) single-chip multiprocessors, be it for servers, PCs, or embedded systems. For brevity, the term “multiprocessor” will be used in this thesis to refer to these (heterogeneous) single-chip multiprocessors. Examples include (in alphabetical order) AMD Phenom processors, ARM Cortex A9, Atmel Diopsis 940, Intel Core processors, Sony/Toshiba/IBM Cell Broadband Engine, Texas Instruments OMAP 3530, and Tilera TILE64, see [BDM09] for a survey. While the shift towards multiprocessors is clear, a tremendous amount of diversity concerning the architectural choices exists. Among others, these choices involve the number and type of processors integrated on one chip, the memory hierarchy and memory consistency model, and the on-chip interconnect. Further choices involve the power management, packaging, and cooling, for instance.

Even though technological constraints play an important role in hardware design decisions, the ultimate yardstick is how well applications can exploit the offered performance.

1.2 Multiprocessor Streaming Applications

From a software viewpoint, harnessing the power of multiprocessors requires software design to move to parallel programming. Until recently, however, parallel programming has mainly been applied to a rather restricted group of applications and architectures, such as scientific and high-performance computing applications, large-scale databases, and data mining applications that typically execute on parallel (super-) computers. Thus, parallel programming has traditionally been a domain for a relatively small group of scientists and engineers whereas the majority of programmers have been writing sequential programs. While microprocessor design is already smoothly migrating to multiprocessors, multiprocessor software design seems to lag a step behind. With the advent of multiprocessors, this problem has become acute and the past years have seen lively discussions on many aspects of multiprocessor software design, such as automated parallelization, programming models, run-time environments, performance analysis, testing, and debugging, to name a few.
One specific class of applications that has attracted particular interest due to its widespread use is the class of streaming applications. Streaming applications can be defined as applications that are centered around the processing of data streams. This class of applications has become commonplace in our daily lives: examples include video, audio, and speech processing, signal processing applications, network packet processing, or (de)compression and cryptographic algorithms. More generally, many elementary mathematical operations, such as matrix multiplications or fast Fourier transforms that underlie mobile telephony and digital television, for instance, can be expressed as streaming applications.

A key characteristic of this class of applications is that data are processed only for a limited amount of time: as opposed to manipulating a (more or less) fixed data set, as in desktop applications, streaming applications receive a stream of data items from an external source, process them, and finally forward them to an output device. Another characteristic is that the data processing can be usually split up into almost independent processing stages on single data items, such as signal samples, audio frames, video frames, or data packets. Due to these characteristics, streaming applications can usually be easily expressed in a parallel fashion and are well suited for execution on multiprocessors, as sketched in Fig. 1.

At the same time, these characteristics can be leveraged for developing design automation tools for multiprocessor streaming applications. This approach is commonly referred to as model-driven development [Sel03] or model-driven engineering [Sch06].

![Figure 1: Streaming applications expressed as process networks (top) mapped onto a multiprocessor (bottom).](image)
1.3 Model-Driven Development

The basic idea behind model-driven development is that developers specify software according to a high-level model that formalizes the essential characteristics of a specific class of applications. In the case of streaming applications, this could be the separation of an application into processes and their interconnection or the size of the data items communicated between processes. By exploiting the properties of the model, a concrete platform-dependent implementation can be automatically generated, creating solutions that are (almost) as good as manually created ones. In short, the premise of model-driven development is to raise the abstraction level and degree of automation in order to increase quality and productivity. Developers can thus concentrate on dealing with issues of essential complexity without being immediately immersed in the details of implementation and accidental complexity. Therefore, this thesis advocates to employ model-driven development for developing multiprocessor streaming applications.

Design flows for model-driven development of multiprocessor streaming applications have been developed even before the occurrence of single-chip multiprocessors. The main domain of these design flows were embedded systems where manufacturers integrated multiple processors or microcontrollers on a single circuit board to realize heterogeneous platforms that possess both good performance and high energy-efficiency. Programming these platforms and multiprocessors is similar because both require device-level programming to a certain degree. Consequently, the demand for automating the design of these systems is high, as well. Specifically, design flows for model-driven development [Pim08, ET06, TKA02] have already been successfully applied to automate software design and performance analysis, demonstrating the potential benefits on concrete case studies.

Unfortunately, the concrete techniques that have been proposed have lost some of their significance because of changed boundary conditions: some of today’s multiprocessors integrate already tens of processors, surpassing the relatively few processors typically integrated on a circuit board in the past. At the same time, these multiprocessors (and the single processors therein) are frequently rather limited in terms of data and instruction memory, interconnect bandwidth, and auxiliary resources, such as timers, or interrupt controllers. In addition, technological evolution opened new directions which are insufficiently addressed in those existing design flows, such as networks-on-chip, 3D-integration, thermal-aware, or process-variation-aware computing.

Thus, making model-driven development applicable to multiprocessors requires the reconsideration of multiprocessor design challenges.
and the development of suitable techniques to address them. Some of these challenges that will be addressed in this thesis are listed below, categorized into software design and performance analysis.

1.3.1 Software Design

Model-driven development opens a gap between the system-level specification and the actual implementation of the design, sometimes referred to as the implementation gap [MK03]. The challenge in bridging this gap is to preserve the system-level semantics on the one hand and achieve the desired performance on the other. This challenge needs to be addressed at different levels:

**Programming Model.** According to [SPRK04, ST98], a programming model is the programmer’s view of an architecture that abstracts the underlying hardware but still exposes enough details to allow efficient implementations. In this sense, a programming model could be anything from a set of system-level API primitives to a particular instance of a model of computation, or a programming language. The challenge is to define a programming model that is general enough to match a certain range of applications, allows efficient execution of applications on a certain range of architectures, and is restricted enough to be amenable to analysis and, thus, automation techniques.

**Run-Time Environment.** The run-time environment is the software that serves as the interface between the hardware and the applications. Its main purpose is to manage and provide access to shared resources, such as processors, interconnects, and memories. In the context of (heterogeneous) multiprocessors and model-driven development, run-time environments tailored to the specific constraints of the programming model and the target architecture are needed to keep the implementation gap small. Particular attention needs to be paid to minimizing the overhead for the run-time environment in terms of both clock cycles and memory consumption. Further challenges relate to memories which are frequently rather small and the memory hierarchy that needs to be explicitly managed by software in many multiprocessors. Generally, these issues are insufficiently addressed in existing run-time environments and embedded real-time operating systems.

**Software Synthesis.** The system-level specification provided by a developer when following a model-driven development approach does not fully define the device-level implementation. Rather, it leaves open implementation details that need to be refined later in the design cycle.
This refinement step that fills the implementation gap is referred to as software synthesis or model compilation [ELLSV97]. Hence, given a system-level specification, the aim of software synthesis is to create a concrete implementation of this specification. Besides attaining functional correctness, software synthesis frequently also encompasses optimizing the performance or memory usage of the implementation.

Depending on the scope of software synthesis, the challenges range from source-to-source code transformations to generating code and solving various optimization problems.

### 1.3.2 Performance Analysis

Performance analysis refers to analyzing timing properties like throughput, response time, delays, or resource utilization and plays a critical role in multiprocessor embedded system design: on the one hand, performance analysis provides the quantitative basis for (automated) design space exploration, that is, for finding a mapping of a given application onto a given multiprocessor, optimizing given objectives. On the other hand, performance analysis is required even for single designs to verify whether their timing constraints are met.

While designing multiprocessors in hardware and software is challenging, this holds equally for performance analysis. Hence, much effort is spent on improving the capabilities of analytic performance analysis methods which have favorable properties such as short analysis times and do not rely on complete implementations. In the context of real-time streaming applications, examples are best-case/worst-case analysis methods such as modular performance analysis (MPA) [CKT03] and symbolic timing analysis for systems (SYMTA/S) [HH1705]. Integrating these methods into a multiprocessor design flow, however, raises many new questions at different levels:

**Scope of Analysis.** Already the performance analysis of uniprocessors is challenging, be it due to speculative components like caches and branch prediction units, irregular behavior of software, or, more generally, incomplete knowledge about implementation details of software or hardware components. This problem is aggravated in multiprocessors that integrate multiple, more or less autonomous processors and interconnect resources that operate in a coordinated manner to reach a common goal. Consequently, there is a strong need for improving existing performance methods for analyzing multiprocessors.

**Analysis Model Generation.** One reason preventing the wide adoption of state-of-the-art performance analysis methods like MPA
and SymTA/S might be the difficulty of creating a faithful performance analysis model: first, creating a model requires familiarity with the analysis method and its theoretical and practical foundations. Second, to be able to model a hardware/software system, deep knowledge about the system itself is required. Third, the modeling effort increases as analysis methods get more powerful. The challenge in model-driven development is to bundle the required knowledge into a tool and automate the model generation.

**Model Calibration.** The more complex systems and performance analysis methods get, the more parameters are associated to a performance analysis model. Consequently, model calibration, that is, obtaining values for these parameters, becomes a time-consuming effort. Model calibration is further aggravated by the need to gather data from different sources, such as data sheets, developer inputs, and executable specifications. Furthermore, determining a parameter value sometimes requires some processing itself. A typical example is the determination of the worst-case execution time of a software task, which is an ongoing research problem on its own \[\text{WEE'08}\].

### 1.4 Contributions

This thesis advocates to use a model-driven development approach to construct multiprocessor streaming applications, mainly because a model-driven development approach allows to automate design tasks which otherwise would need to be done manually. This does not only reduce design time and costs but also enables software developers to use techniques without needing to acquire the domain knowledge underlying these techniques. Specifically, this is demonstrated for two design phases, namely software design and performance analysis. First, it is shown that an efficient platform-dependent implementation of a streaming application can be generated from a platform-independent application specification. The software developer is thus relieved of getting familiar with the device-level details of a multiprocessor. Second, it is shown that performance analysis for real-time multiprocessor streaming applications can be automated. Therefore, software developers can leverage the benefits of a state-of-the-art performance analysis method without a deep knowledge of the underlying mathematical details.

Central to model-driven development is a programming model. In this thesis, the dataflow process network \[\text{Kah74, LP95}\] model of computation is employed. Basically, a dataflow process network consists of parallel processes that communicate via point-to-point first-
in first-out (FIFO) channels, as indicated in Fig. 1. The dataflow process network semantics not only matches well the characteristics of streaming applications, it is also a solid basis for efficient and scalable multiprocessor implementations: the dataflow model is an untimed model of computation, that is, the functional input/output relation is independent of the timing of the execution, and the model explicitly separates computation from communication.

Focusing on the dataflow model of computation, the following contributions are made in this thesis:

**Lightweight Multiprocessor Run-Time Environment.** Whether model-driven development approaches will finally be adopted on a larger scale, depends decisively on the actual performance of applications when executed on multiprocessors. The goal is that an implementation that is automatically derived from a high-level specification executes as efficiently as a manually created one. Taking speed-up as a metric for efficiency, that is, the ratio of uniprocessor versus multiprocessor execution time, it is shown that this is actually possible for dataflow process networks. In this context, the challenge is to keep the overhead for executing a process network to a minimum where context switches between processes and FIFO channel accesses contribute the biggest share to this overhead. The proposed solution is a multiprocessor run-time environment that is based on so-called protothreads [DSVA06] to implement the quasi-parallelism required for executing multiple processes on a single processor. Moreover, so-called windowed FIFOs are used instead of standard FIFOs for efficient FIFO communication. Compared to existing run-time environments, the proposed run-time environment is lightweight in the sense that it has a reduced overhead in terms of processor cycle count and a smaller memory footprint. Furthermore, it does not rely on (inline) assembly code but is exclusively written in standard ANSI C, making it easy to port to different multiprocessors. In particular, the run-time environment has been ported to the Sony/Toshiba/IBM Cell Broadband Engine [PAB+06], a heterogeneous multiprocessor integrating nine cores. In addition, a software synthesis tool for executing process networks on top of this run-time environment is implemented. It is shown that this way a speed-up can be achieved that is almost linear in the number of processors without resorting to device-level programming.

**Performance Analysis of Processes with Complex Activation Schemes.** Single processes in streaming applications often process data iteratively and in a piecewise fashion from multiple input streams. An activation scheme describes how many data pieces need to be available
on each of these input streams so that a process can enter the next iteration. A faithful performance analysis of such processes needs to take this activation scheme into consideration. When considering best-case/worst-case performance analysis methods, mainly simple activation schemes like OR- and AND-activation have been considered, that is, processes are activated when data are available on any input stream, or all input streams, respectively. In realistic systems, however, more complex activation schemes are frequently encountered: processes might read from input streams with different rates, the rates might be variable, and, more generally, the activation scheme might depend on the state of the process. Dealing with this challenge, first a rather general process model is introduced. For processes adhering to this model, subsequently a best-case/worst-case performance analysis method is proposed. In particular, the method uses the framework of real-time calculus [TCN00] and is integrated into the framework of MPA, extending its scope to applications exhibiting complex activation schemes.

Automated Generation and Calibration of Compositional Performance Analysis Models. Performance analysis is required throughout the multiprocessor design flow, starting from (early) design space exploration to final system verification. Therefore, it is proposed to seamlessly integrate automated performance analysis into a model-driven development approach to relieve the system developer or performance analyst of the demanding and time-consuming task of manually carrying out the performance analysis. The proposed approach is split up into a model calibration phase during which all data is collected and annotated to the system specification and a model generation phase during which an analytic system model is generated based on the annotated system specification. Focusing on real-time streaming applications, MPA is used as the performance analysis method. Compared to related work in which mostly simulation-based approaches and average-case analysis methods are used, MPA provides full corner-case coverage and is thus applicable for the design of hard real-time systems. The benefits and limitations of the approach are investigated by performing experiments on a multiprocessor ARM platform [BBB05].

Integration into the Distributed Operation Layer. The distributed operation layer (DoL) is a framework for model-driven development based on the dataflow model of computation. As such, the DoL provides a framework for specifying applications, an application programming interface (API), and automated software synthesis for uniprocessors and multiprocessors. Being the main contribution of the Computer Engineering Lab at ETH Zurich to a 48-months European research
project entitled “Scalable Software/Hardware Architecture Platform for Embedded Systems” [PJL+06], a group of people contributed to these basic features of the DoI. By additionally integrating the contributions proposed above into the DoI, its scope is extended in two ways: first, the range of supported architectures is extended to the Sony/Toshiba/IBM Cell Broadband Engine. Second, by integrating the proposed methods for model generation and calibration, the scope of DoI is extended to the design of real-time systems.

Parts of the work presented in this thesis have been published in the following book chapters, journal articles, and conference papers: [TBHH07] (Chapter 2), [HSH+09] (Chapter 3), [HT07] (Chapter 4), [HKH+09, HBLH09, SBF+07] (Chapter 5), [HHBT09a, TWH09, BHHT10] (all chapters). The DoI itself is available online for download at http://www.tik.ee.ethz.ch/~shapes/.

1.5 Thesis Outline

In Chapter 2, the distributed operation layer design flow is introduced. The typical architectural features of multiprocessors are reviewed which leads to four themes that need to be considered when developing multiprocessor streaming applications: efficiency, scalability, portability, and predictability. The general approach to address these themes by making use of dataflow process networks is presented and compared to related frameworks.

In Chapter 3, the themes of efficiency, scalability, and portability are addressed by proposing a run-time environment for executing dataflow process networks on multiprocessors. After introducing the general approach based on so-called protothreads and windowed FIFOs, a prototype implementation for the Sony/Toshiba/IBM Cell Broadband Engine is described. The efficiency, scalability, and portability of the run-time environment are verified by means of a series of experiments.

In Chapter 4, the predictability of dataflow process networks is addressed by considering best-case/worst-case compositional performance analysis methods to predict their real-time behavior. In this context, a method is proposed which allows to analyze actors with multiple inputs and outputs whose activation can be described by OR- and AND-activation or combinations, thereof. The results obtained by using this method are compared with results reported in the related literature.
In Chapter 5, the focus is on the automated generation and calibration of compositional performance analysis models for multiprocessor streaming applications. The chapter introduces a general approach which is then set out in detail in the context of the DoL and MPA. The viability of the approach is assessed by integrating a prototype implementation into the DoL and carrying out automated performance analyses for several applications.

Finally, Chapter 6 presents a summary of the contributions and some concluding remarks.

Supplementary material for Chapters 2, 3, and 4 is collected in the Appendix which contains the source code and several illustrations for a concrete multiprocessor streaming application, illustrating the different design steps.
The distributed operation layer (Dol) is a design flow for the model-driven development [Sel03] of multiprocessor streaming applications and has been developed at the Computer Engineering Group at ETH Zurich in the context of a European Framework Programme 6 research project called “Scalable Software/Hardware Architecture Platform for Embedded Systems” (SHAPES) [PJL+06]. From January 2006 to December 2009, SHAPES dealt with the hardware/software codesign of a tiled multiprocessor. Exploiting the raw performance of the SHAPES multiprocessor without resorting to time-consuming device-level programming was the main challenge from a software perspective. (This applies to other multiprocessors as well.) At the task level, this requires hardware units or compilers that can extract the instruction-level parallelism of sequential program code to exploit multiple logic and arithmetic units found in today’s superscalar and very large instruction word (VLIW) processors. At the application level, exploiting performance requires leveraging task-level parallelism to be able to distribute the computation load across the available cores. Assuming that instruction-level parallelism can be well handled using already existing techniques and tools, the Dol was conceived as a programming framework dealing with questions at the application level. As such, the Dol is supposed to aid software developers in specifying parallel streaming applications and implementing them on multiprocessors.

This chapter introduces the Dol as the design flow that is the basis of this thesis and the design flow into which the proposed methods have been integrated. First, the architectural characteristics of the class of considered multiprocessors are described. On the basis of these characteristics, the requirements for the Dol as a programming
framework are determined. This is followed by a description of the programming model underlying the Dol, which is the basis for satisfying these requirements, and a description of the Dol design flow. The chapter concludes with a discussion of design flows related to the Dol and a short summary.

2.1 Target Architectures

Even though the Dol was developed in the context of the SHAPES project, it was designed to support a broader range of multiprocessors from the beginning. These multiprocessors can be characterized as follows:

**Scalable, Heterogeneous Architecture.** Heterogeneous multiprocessors integrate different types of processors that can efficiently execute different types of workloads. Frequently, reduced instruction set computer (RISC) processors are combined with digital signal processors (DSPs) to create an architecture that is well suited for control-oriented and numerical computations alike. A central aspect of (heterogeneous) multiprocessors is scalability which can be interpreted in different ways: on the one hand, the number of cores increases from one processor generation to the next. On the other hand, multiprocessors belonging to the same generation of a processor family might have a different number of cores but otherwise the same features. Furthermore, multiple multiprocessors might be integrated into bigger systems. Therefore, scalability needs to be considered when designing multiprocessor hardware components and application software.

**Distributed Memory Architecture.** Heterogeneous multiprocessors often have a distributed memory architecture in which individual processors have their own local memory. In these architectures, the data transfer between memories needs to be explicitly managed by software via message-passing. This allows the use of optimized policies for carrying out these transfers but creates trade-offs in performance, code complexity, and optimization effort. This is in contrast to multiprocessors with coherent hardware-managed caches providing the abstraction of a single shared address space. Note that as compared to off-chip memories with typical capacities in the range of MBytes or even GBytes, the on-chip memories of multiprocessors are usually rather small with typical capacities in the range of kBytes.
Advanced Interconnect. The integration of multiple processors on a single chip requires advanced interconnects because a single shared bus usually cannot provide the required quality of service. Solutions to this range from using multi-layer buses to networks-on-chip with different topologies. In addition, usually different types of controllers (direct memory access controllers, memory flow controllers, routers, and (distributed) network processors) form an integral part of these interconnects. The controllers can carry out data transfers independent of the processors, allowing the overlap of computation and communication. Furthermore, interconnects frequently have a hierarchical structure with different types of interconnects used at different levels. Unlike uniprocessors, in which interconnects are seldom considered at the application level, in multiprocessors, interconnects might have a significant impact on performance. In particular, the size of messages, size and location of communication buffers, and sometimes the routing and scheduling of messages need to be taken into account.

One example for an architecture with these characteristics is the multiprocessor designed in the SHAPES project, see Fig. 2. This architecture is based on a heterogeneous tile that integrates an ARM 9 RISC processor and an Atmel mAgicV floating point VLIW DSP. The SHAPES tile has a distributed memory architecture with two separate on-chip memories tightly coupled to the ARM 9 and mAgicV processors. This SHAPES tile was actually manufactured and commercialized as Atmel Diopsis 940 (device number: AT572D940HF), which has a peak performance of roughly $10^9$ floating point operations per second at a clock frequency of 100 MHz and a power consumption of approximately 0.5 W. Multiple tiles can be integrated into large multiprocessors, with SHAPES focusing on an architecture with eight tiles. This multi-tile architecture features a hierarchical interconnect in which a multi-layer advanced high-performance bus (AHB) connects the hardware components on each tile, whereas different tiles can communicate with each other via a network-on-chip.

Besides the Atmel Diopsis 940, the Sony/Toshiba/IBM Cell Broadband Engine and the Texas Instruments OMAP 3550 are multiprocessors showing these characteristics, for instance. Applications specified according to the DoJ specification format can currently be automatically ported to the following multiprocessors:

- Atmel Diopsis 940 [Pjl+06]
- Multiprocessor ARM Platform (Mparm) [BBB+05]
- Sony/Toshiba/IBM Cell Broadband Engine [PAB+06]
Figure 2: Block diagram of a single tile (left) and the multi-tile SHAPES architecture (right). The availability of six links on the distributed network processor enables the construction of a scalable on/off-chip toroidal network.

Note that even though homogeneous multiprocessors and (general-purpose) graphics processing units [ND10] may share some of these characteristics, they are quite different in that at least a group of cores usually share a common memory and hardware multi-threading is supported. In this case, communication solely via message-passing is inefficient, making the DoI unsuitable because of its focus on communication via message-passing.

2.2 Efficiency, Scalability, Portability, and Predictability

Considering the class of architectures characterized above, the following requirements have been identified for (real-time) streaming applications executing on these architectures.

Efficiency. The efficiency of an application depends on the application code itself, its mapping onto a given multiprocessor, and the run-time environment. In general, the goal is to distribute the computation and communication load uniformly across the multiprocessor, and possibly overlap computation and communication to hide communication latencies. Ultimately, high efficiency means that the available computation and communication resources of a multiprocessor are well utilized, which can, for instance, be measured in terms of speed-up or the achieved peak performance.
2.2. Efficiency, Scalability, Portability, and Predictability

**Scalability.** Like multiprocessors also streaming applications are scalable in various aspects, such as problem complexity and size, functional and data parallelism, or task granularity. For example, in a video encoding algorithm, different processing steps can be carried out at the frame, slice, macroblock, or block level which influences the task granularity and the manner in which the functional and data parallelism is exploited. Furthermore, the compression rate, frame rate, and resolution, for instance, are parameters that determine the actual problem complexity and size. As scalability is a characteristic property of streaming applications, it is desirable to parameterize these applications such that they can be easily configured for multiprocessor architectures of different scales.

**Portability.** Portability means that the implementation of an application does not contain any architecture-specific code that limits its execution to only one architecture. In the context of multiprocessors, portability is desirable not only to enable the porting of an application to different architectures but also to facilitate moving single processes from one processor to another on the same multiprocessor. The reason is that the mapping of an application onto a heterogeneous multiprocessor is usually not known from the beginning and, therefore, it would be time consuming to manually modify the code whenever the mapping changes. The same reasoning applies to systems in which processes can dynamically migrate from one processor to another during run-time. Furthermore, parallel applications or, at least, individual processes are frequently developed and functionally tested on uniprocessors and, consequently, they should be executable on uniprocessors, as well.

**Predictability.** In this thesis, the term predictability refers to the property of an application that its timing behavior can be quantitatively estimated or bounded without actually running or simulating it. In this sense, predictability means that an application is amenable to a formal or an analytic performance analysis method. In particular, with respect to real-time streaming applications, predictability is required to be able to assert whether a system meets the given real-time constraints. More generally, predictability is also desirable for design space exploration: evaluating the performance of a single point in the design space takes tens of seconds when being based on measurements or simulation because of the time required for compiling multiple binaries and executing them on a real or virtual (simulation) platform. During design space exploration, however, typically thousands of design points need to be evaluated, which becomes prohibitively time consuming when only based on measurements or simulations. Consequently, predictability and
fast performance analysis methods are required to carry out design space exploration within a practical timeframe.

2.3 Programming Model

The characteristics of multiprocessors and the resulting requirements for software, as discussed in the previous two sections, render multiprocessor software development a difficult task. Unfortunately, there is currently little support for software developers to carry out this task: despite the existence of promising approaches, see the overviews in [wPOH09, KB09], (de-facto) standards still need to be established to lay a foundation for the development of mature design automation tools. Meanwhile, the typical practice is to program multiprocessors in a conventional sequential language, such as C/C++, at device-level based on a so-called board support package and custom development environments provided by the multiprocessor hardware manufacturers. While this approach offers considerable flexibility to software developers and (potentially) enables efficient implementations, the resulting software hardly satisfies the requirements of scalability, portability, and predictability, not to speak of being a rather time-consuming approach.

To overcome these serious drawbacks of a manual approach, the Dol is based on the concept of model-driven development, which restricts the implementation of applications by a programming model that imparts the desired properties. Hence, applications become tractable, which allows automating design steps that are beyond the reach of compilers for conventional programming languages. The resulting loss of flexibility is then outweighed by the possibility to automatically generate platform-dependent implementations and automatically carry out performance analysis and design space exploration. Being at the core of any model-driven development approach, this section presents the programming model upon which the Dol is based.

Briefly, the programming model of the Dol can be described as a concrete instance of the dataflow process network model of computation [LP95], a subclass of Kahn process networks [Kah74]. Basically, a dataflow process network expresses an application as a set of parallel and autonomous processes that communicate exclusively via point-to-point first-in first-out (FIFO) channels. Executed (firing) repeatedly, processes in a dataflow process network map atomic data objects (tokens) from one or more input streams to one or more output streams. Note that the processes are commonly referred to as actors in the context of dataflow process networks.
2.3. Programming Model

The process network model used in the Dol. differs from the original dataflow process network model, as introduced in [LP95], in one aspect, namely the finite capacity of channels: basically, the FIFO channels in a dataflow process network have unbounded capacity. Since unbounded channels cannot be realized in physical implementations, the Dol. is based on channels with finite capacity. As long as the insufficient capacity of a channel does not cause a deadlock, however, this does not imply a semantic change. Unfortunately, preventing these so-called artificial deadlocks that would not occur in a dataflow process network with unbounded channels is not possible at design time because of the Turing completeness of dataflow process networks. Therefore, the occurrence of artificial deadlocks cannot be generally prevented in the Dol. Nevertheless, the experience is that most applications exhibit a regular communication behavior that allows the software developer to quantify the capacity of FIFO channels such that artificial deadlocks do not occur. Besides analysis at compile time for restricted communication behavior, another possibility to deal with artificial deadlocks are run-time approaches that detect and resolve deadlocks during execution [Par95].

2.3.1 Application Programming Interface

The dataflow model can be seen as a coordination model [GC92] that allows to consider the development of a parallel application as the combination of computation and coordination: in the context of dataflow process networks, the developer describes computation by implementing individual, sequential actors that manipulate data streams and coordination by the connection of actors using channels. This separation of concerns is also reflected in the Dol. application programming interface because two different languages are used for the two purposes.

Computation-wise, standard C/C++ is used as the host language to implement actors. This mainly allows to reuse already existing legacy code and to execute actors on any processor for which a C/C++ compiler is available. Specifically, the application developer has to specify each actor, as follows:

- A custom data structure stores the local state of an actor.
- INIT() initializes the local state of an actor.
- FIRE() specifies the actual function of an actor that is repeatedly executed.
- Finally, actors can invoke DETACH() to indicate completion, allowing the termination of a process network as soon as all the actors stop.
For communication, actors can access (local) ports that serve as the interface to the FIFO channels using blocking read() and write() primitives. Blocking means that these primitives are stalled until the specified number of tokens is available in the FIFO channel. Furthermore, note that in a standard implementation, there are three copies of each token communicated over the FIFO channel: one at the sender, one in the FIFO channel buffer, and one at the receiver. write() copies a token from the local memory of the sender into the FIFO buffer wherefrom read() copies the token to the local memory of the receiver.

Coordination-wise, the extended markup language (XML) is used as the coordination language for describing the topology of a dataflow process network, that is, the instantiation of actors and their connection by the FIFO channels. The topology is thus not hard-coded in the C/C++ implementation, which allows the rapid creation and modification of process networks. To facilitate the creation of large, scalable process networks, the XML-based description allows the use of so-called “iterators”. Similar to the “generate” statement in VHDL, iterators can be used to describe the regularly structured parts of a process network, such as split-and-merge structures or mesh and butterfly topologies.

Note that the application specification is abstract in the sense that neither the FIFO implementation nor the scheduling of actors or the mechanism for their parallel execution is specified. Refining these details is deliberately left to the software synthesis step, keeping the application specification itself completely platform-independent. Examples for an XML file and the source code of an actor are listed in the Appendix A in Section A.1.

2.3.2 Modeling Multiprocessor Streaming Applications as Dataflow Process Networks

The process network model matches well with the algorithmic structure of many streaming applications. In particular, the decomposition of an application into functionally independent processing stages and the piecewise processing of data items that is characteristic of streaming applications can be naturally modeled by dataflow process networks. In addition, the dataflow model explicitly separates computation from communication, which facilitates implementations on the targeted class of multiprocessors. In this manner, the dataflow process network model lays the foundation for efficiency, scalability, portability, and predictability, as follows:
Efficiency. With respect to multiprocessors, the dataflow model is determinate, which allows executing process networks in an untimed (asynchronous, data-driven) fashion. Since it requires no global coordination or synchronization, untimed execution also imposes fewer constraints on the multiprocessor hardware and run-time environment support as compared to time-triggered execution \cite{LVB09}. Hence, dataflow process networks can be efficiently executed on a wide range of platforms with different processor, interconnect, and memory configurations.

Scalability. Due to the encapsulation of actors that only operate on local data and communicate via point-to-point channels accessed by (local) ports, dataflow process networks are modular and, thus, scalable. Specifically, dataflow process networks can be executed in a distributed manner, that is, without any “master” with global knowledge that could easily become the bottleneck of a system.

Portability. Only specifying the topology of the process network and the semantics of the communication API routines allows mapping a dataflow process network onto any platform that provides an appropriate run-time environment. Such a run-time environment essentially has to provide a mechanism for running multiple actors in a quasi-parallel manner on a single processor and a FIFO channel implementation.

Efficiency, scalability, and portability are considered in detail in Chapter 3.

Predictability. The formal semantics of dataflow process networks facilitate their formal or analytic performance analysis. In the domain of real-time systems, typical abstractions used in performance analysis, such as event streams and tasks, have corresponding counterparts in the dataflow model of computation, namely data streams and actors. Therefore, many powerful abstractions that have been developed to model and analyze distributed systems, see \cite{PWT09} for a comparison of multiple methods, are conceptually applicable to multiprocessor streaming applications modeled as dataflow process networks. Predictability could be further increased by restricting to a subclass of dataflow process networks, such as synchronous dataflow \cite{LM87}, at the cost of losing expressiveness.

The performance analysis of dataflow process networks in the context of model-driven development is considered in Chapters 4 and 5.
2.4 DoL Software Design Flow

Programming uniprocessors or the device-level programming of multiprocessors usually follows a familiar pattern: software developers write code in a sequential language, compile it, and link it against the platform-dependent libraries to create an executable. Hence, apart from using a development environment that facilitates the creation of code and a compiler tool-chain, the degree of automation is small. This can be changed by employing a model-driven development approach, as advocated in this thesis, because specifying an application at a high abstraction level increases the dependence of software developers on design automation tools. Obviously, the concrete steps, tools, and the involvement of the developer differ between design flows because of the different application domains, scopes, and techniques used. Therefore, this section informally describes the DoL design flow as it does not closely follow any standardized multiprocessor software design flow either.

2.4.1 Y-Chart Approach

The DoL software design flow basically follows the so-called Y-chart approach, as shown in Fig. 3. Kienhuis et al. [KDVvdW97; KDvdWV02] proposed the Y-chart approach for the hardware/software codesign of programmable multiprocessors as a framework to collect the quantitative data in order to analyze the performance of a system and provide the basis for taking design decisions.

![Y-chart approach for designing multiprocessors and multiprocessor software.](image)

The principle underlying the Y-chart approach is the orthogonalization of concerns [KMN00], in the sense that application, architecture, and mapping specifications are separated. Maintaining this separation until the latter stages of the design cycle allows the easy modification of either of these specifications and iterative improvement a system. In the case of the DoL, the Y-chart approach has been adopted as follows:
2.4.1 Application Specification and Functional Simulation

The starting point of the Dol software design flow is the application specification. During application specification, the software developer
specifies the application as a dataflow process network according to the programming model introduced in Section 2.3, that is, using XML to define the topology of the process network and C/C++ to implement the behavior of individual actors. Since process networks have an intuitive graphical representation as graphs where nodes represent actors and edges represent FIFO channels, a graphical editor might be used to specify the topology of the process network. In the case of the DOL, a plug-in for the modeling, simulation, and evaluation of systems (Moses) [Jan00] graphical editor has been developed for this purpose, see Fig. 4 in the Appendix A.1.5. This plug-in also allows the creation of C/C++ code templates for the actors as a starting point for software development.

Due to the high abstraction level, however, the application specification cannot be directly compiled into an executable because it requires a run-time environment that provides a mechanism to run multiple actors in a quasi-parallel fashion and a FIFO channel implementation for communication. To execute an application on a standard (uniprocessor) workstation, the DOL provides a run-time environment and a software synthesis tool to create an executable based on the application specification and the run-time environment. As the execution on a uniprocessor mainly gives an insight into functional properties rather than non-functional ones, this executable is also referred

**Figure 4:** Overview of the DOL software design flow.
to as a functional simulation. Using the functional simulation, an application can be functionally tested and debugged using a standard compilation tool-chain and debugging tools, such as a standard C/C++ compiler (gcc/g++) and the GNU debugger (gdb).

Technically, the DOL run-time environment for uniprocessors is based on existing thread libraries. Specifically, POSIX threads (pthreads) [IEE04] and SystemC threads (QuickThreads) [Kep93] have been evaluated where QuickThreads have an advantage in terms of execution speed because they are user-space threads rather than kernel-space threads like pthreads. Based on QuickThreads and a custom FIFO implementation, the creation of a functional simulation involves the following steps, resulting in the structure shown in Fig. 5.

- Parse the XML file to determine the topology of the process network.
- Create a main file for bootstrapping the process network, as follows:
  - For each actor, create a thread that first invokes the actor’s init() routine and then repeatedly the fire() routine.
  - Instantiate FIFO channels and establish a pointer between each actor’s ports and the corresponding FIFO channels.
  - Run all threads.
- Create a directory structure and a Makefile to facilitate the compilation and build process of the functional simulation.

Finally, note that a multi-threaded functional simulation can exploit multiprocessors that (transparently to the programmer) run multiple threads in parallel. Today’s workstations typically have this capability, for instance, being equipped with multicore processors running an operating system with support for symmetric multiprocessing. In this case, the “functional simulation” actually becomes a multiprocessor implementation that (potentially) executes faster than its sequential (uniprocessor) counterpart, as reported in [EVT08] for pthreads, for instance. Focusing on target architectures with a distributed memory hierarchy, this direction was not further investigated in the context of the DOL, however.

### 2.4.3 Architecture and Mapping Specifications

The architecture and mapping specifications in the DOL serve two purposes. On the one hand, these specifications are inputs to the software synthesis tool. On the other hand, they are the basis for the performance analysis. Examples for both architecture and mapping specifications,
which are stored in an XML-based format, are listed in the Appendix A in Section A.1.

With respect to software synthesis, the specifications must contain the configuration information required by the multiprocessor software synthesis tools. In the case of the DoL, this is the binding of actors to processors, the binding of FIFO channels to communication drivers, and the scheduling of processors (scheduling policy and parameters).

With respect to performance analysis, the architecture and mapping specifications have to capture the relevant properties of the entire hardware/software system (in connection with the application specification), where the required specific information depends on the employed analysis method. Therefore, usually additional information is needed than just for software synthesis. As an example, a high-level description of the microarchitecture of the multiprocessor and the associated parameters, such as processor frequencies and interconnect bandwidths, might be defined in the architecture specification. Similarly, the mapping specification might indicate the architectural components to which a communication driver corresponds to, for instance.
2.4.4 Multiprocessor Run-Time Environment and Software Synthesis

A multiprocessor run-time environment for process networks basically has to provide a mechanism to run multiple actors in a quasi-parallel fashion and FIFO implementations for communication between actors located on the same and different processors. The following run-time environments providing these services have been considered on the different multiprocessors.

**Atmel Diopsis 940.** In SHAPES, a custom multiprocessor operating system for the Atmel Diopsis 940 was developed from scratch by a project partner [GPY07, GP09]. To effectively address the challenges related to the heterogeneity and scalability of multiprocessors, this operating system called DNA-OS (DNA is not just another operating system) has a component-based architecture. In DNA-OS, the (micro-) kernel provides only a very small set of elementary services (thread management, synchronization primitives, and basic scheduling), whereas other services (dynamic memory management, file system support, and device drivers) are implemented as components that can be optionally added.

**MPARM.** Focusing on the execution of real-time applications, a port of the real-time executive for multiprocessor systems (RTEMS) operating system [RTE10] is available for MPARM. RTEMS provides a full-fledged preemptive scheduler and message queues such that process networks can be directly implemented on top of it [HHBT09b].

**Cell Broadband Engine.** A run-time environment for process networks on the Cell Broadband Engine is one contribution of this thesis, see Chapter 3. With an emphasis on efficiency in terms of run-time overhead and memory usage, this run-time environment is based on a lightweight thread implementation and an efficient FIFO implementation.

Based on the platform-independent application specification and a multiprocessor run-time environment for process networks, the task of software synthesis is the creation of the platform-specific implementation according to the architecture and mapping specifications. Based on the run-time environments described above, the software synthesis steps and the structure of the resulting software stack are basically the same as those for functional simulation. Further details are therefore omitted here.

Concluding this section, the DoL software design flow is a powerful framework for specifying multiprocessor streaming applications at a high
abstraction level as dataflow process networks and automatically refining them to different implementations. Specifically, automatically refining a specification into a functional simulation enables multiprocessor streaming applications to be developed on uniprocessors, similar to sequential applications. As soon as the desired functionality is implemented, a multiprocessor implementation can be automatically generated by software synthesis. Due to the determinism of dataflow process networks, this implementation is guaranteed to be functionally equivalent, assuming that the application code does not contain platform-dependent code and that no artificial deadlocks occur.

2.5 Related Work

Multiprocessor software design flows following a model-driven development approach have received considerable attention in the literature. General surveys on the existing design flows are presented in [DSVP06, GHP+09], for instance, and a survey focusing on design flows based on the process network model of computation is presented in [HHBT09a]. This section focuses on a few design flows closely related to the DOL and their unique features.

**Eclipse and C-Heap.** Eclipse [RvEJ+02] and CPU-controlled heterogeneous architectures for signal processing (C-Heap) [NKG+02] belong to the early attempts of designing multiprocessors using the process network model of computation. Developed at the Philips Research Laboratories, both frameworks propose a template for a multiprocessor hardware/software architecture for embedded systems. Laying more focus on hardware aspects than other design flows, Eclipse and C-Heap propose to use so-called (hardware) “shells” at the interface between processors and the interconnect to simplify the software implementation of FIFO channels. Recently, C-Heap was reused in composable and predictable multi-processor system on chip (CoMPSoC) [HGBH09] with the goal of creating a template for multiprocessors with a predictable worst-case behavior.

**Artemis and Espam.** The design flows of Artemis [Pim08] and embedded system-level platform synthesis and application mapping (Espam) [NSD08] are similar in their scope, targeting the automated generation of synthesizable register-transfer level specifications for multiprocessors, starting from sequential programs. Several modules of both design flows have been combined in the Daedalus design flow [NTS+08].
Compared to other related design flows, **ARTEMIS** and **ESPAM** feature a front-end that allows to derive a process network specification from a restricted set of sequential applications. In **ARTEMIS**, the **COMPAAN** [KRD00] tool is used to derive a process network from affine nested loop programs written in a subset of Matlab. In **ESPAM**, the process network tool [VNS07] is used to transform applications written as a static affine nested loop program in C into a process network.

In addition to this extension of the front-end of the design flow, **ARTEMIS** and **ESPAM** also feature powerful back-ends for system implementation. In particular, these back-ends allow to generate register-transfer level code for the synthesis of the hardware platform, including processors, interconnects, and dedicated hardware components. Specifically, these back-ends have been used for synthesizing multiprocessors on field programmable gate arrays.

**KOSKI**. The **KOSKI** [KKO+06] design flow is similar in scope to **ARTEMIS** and **ESPAM**, except that the design flow starts from a unified modeling language (UML) specification based on finite state machines instead of a sequential program. Similar to **ARTEMIS** and **ESPAM**, **KOSKI** uses simulations for performance analysis and allows automatic calibration of the simulation based on the implementation and profiling of a small set of designs.

A unique feature of the **KOSKI** design flow is the implementation of an execution monitor that can be connected to the running system. This monitor allows to dynamically map actors in a drag-and-drop fashion onto different processors without recompilation of the software binaries. This enables the designer to explore the design space in an interactive manner.

**STREAMIT** and **MAMPS**. The **STREAMIT** [IKA02] and multi-application multi-processor synthesis (MAMPS) [KFH+08] design flows are both based on the synchronous data flow (SDF) [LM87] model of computation. While this restriction narrows the scope of applications that can be modeled, it opens a wide range of optimization techniques because SDF applications can be statically analyzed.

Specifically, the **STREAMIT** design flow is targeted at exploiting trade-offs between parallelism and the resulting communication overhead, thereby achieving an efficient mapping of an SDF application onto a given architecture. Compared to the other design flows, which assume a fixed structure of the application during design space exploration, the **STREAMIT** design flow allows the automatic exploration of different structures of the process network by actor fusion and fission, that is, by merging and splitting actors. This allows the adjustment of the granularity of actors.
and enables the evaluation of the resulting trade-offs. In addition, the StreamIt design flow allows to derive an optimized static scheduler for an SDF application, which incurs a lower scheduling overhead than event-driven schedulers.

The MAMPS design flow is unique in incorporating techniques to deal with multiple use cases of multiple applications mapped onto a multiprocessor. (A use case or scenario is a specific combination of applications running concurrently over a certain amount of time.) The goal is to minimize the hardware requirements for running a set of use cases.

**OpenDF and Shim.** The OpenDF [BBJ+08] and the software/hardware integration medium (Shim) [ET06] design flows are targeting reconfigurable hardware and multiprocessors. The cores of these design flows are domain-specific languages for dataflow programming. In OpenDF, the CAL actor language [EJ03] is used, whereas Shim uses a subset of C augmented with constructs for concurrency, communication, and exceptions. Recently, the CAL language has also been adopted by the ISO/IEC standardization organization for a new MPEG standard called reconfigurable video coding (ISO/IEC FCD 23001-4, 23002-4). The focus of both frameworks is on techniques for the efficient synthesis of hardware (VHDL and Verilog code) and software (C code).

Besides the aforementioned holistic design flows, there are many tools for supporting individual design steps in a design flow based on process networks. Ptolemy [BLL+08] is an integrated framework for the specification and simulation of applications modeled as dataflow process networks. The dataflow interchange format (DIF) [PSKB08] is a language for specifying dataflow process networks. The language package also includes a translator from DIF to C. A design flow focusing on the software implementation of dataflow process networks expressed in Simulink is presented in [PGR+08]. PEACE [HKL+07] and SystemCoDesigner [KSS+09] are hardware/software codesign flows with PEACE based on a heterogeneous model of computation encompassing finite state machines and SDF and SystemCoDesigner based on an actor-oriented model. Finally, YAPI [dKES+00] is a C/C++ library for programming applications as dataflow process networks and provides a uniprocessor run-time environment for their execution.

Concluding, Table 1 lists several multiprocessor software design flows based on the dataflow model and the targeted multiprocessors. Whenever available, the web links to the design flows are listed.
Table 1: Selected dataflow process network software design flows and examples for targeted multiprocessors.

<table>
<thead>
<tr>
<th>DESIGN FLOW</th>
<th>MULTIPROCESSOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daedalus(^1) [Pim08, NSD08]</td>
<td>multi-MicroBlaze implemented on Xilinx Virtex-II Pro FPGA</td>
</tr>
<tr>
<td>Koski [KKO(^1)+06]</td>
<td>multi-NIOS implemented on Altera Stratix-II FPGA</td>
</tr>
<tr>
<td>Mamps(^2) [KFH(^1)+08]</td>
<td>multi-MicroBlaze implemented on Xilinx Virtex-II Pro FPGA</td>
</tr>
<tr>
<td>Shim [ET06]</td>
<td>Sony/Toshiba/IBM Cell Broadband Engine</td>
</tr>
<tr>
<td>StreamIt(^3) [TKA02]</td>
<td>Sony/Toshiba/IBM Cell Broadband Engine</td>
</tr>
</tbody>
</table>

\(^1\)http://daedalus.liacs.nl
\(^2\)http://www.es.ele.tue.nl/mamps
\(^3\)http://www.cag.lcs.mit.edu/streamit

Finally, note that there are few consolidation efforts between the design flows [PSN\(^1\)+08]. Consequently, no commonly accepted specification format exists, which, in turn, inhibits the interoperability of tools from different design flows. Similarly, there is no widely used benchmark suite, which makes the comparison of design flows difficult. Note that a similar observation currently also applies to multiprocessor software standards, in general, as pointed out in [HAB\(^1\)+09].

2.6 Summary

This chapter introduces the Do\(\)l as a design flow for developing multiprocessor streaming applications. Targeted at scalable, heterogeneous multiprocessors with a distributed memory architecture and advanced interconnects, the Do\(\)l emphasizes efficiency, scalability, portability, and predictability of applications. Following a model-driven development approach, a programming model based on dataflow process networks forms the basis of the entire design flow, thereby imposing the desired properties. The actual software design flow is based on the Y-chart approach, in which the application is specified in a platform-independent manner and explicit architecture and mapping specifications are used to specify a multiprocessor implementation. Automatic software synthesis
is then used for the actual implementation wherein the DoL features software synthesis tools for uniprocessors, the Atmel Diopsis 940, MPARM, and the Sony/Toshiba/IBM Cell Broadband Engine. Hence, the software developer can solely concentrate on issues of essential complexity, being completely relieved from any device-level programming.
One premise of model-driven development is that an application can be specified at a high abstraction level and automatically refined into an efficient implementation. When an application can be adequately expressed in the high-level model, the efficiency of the final implementation depends on how well a design flow can close the implementation gap. In the context of the DoL and dataflow process networks, this depends on the following:

**Multiprocessor Run-Time Environment.** The run-time environment provides the services required to execute a process network, that is, running multiple actors in a quasi-parallel manner and a FIFO implementation.

**Software Synthesis.** Software synthesis creates the link between the application and the application-independent run-time environment. Software synthesis is not further considered in this chapter because it has already been discussed in Chapter 2.

In this chapter, a run-time environment for executing dataflow process networks on multiprocessors is proposed. The run-time environment has been ported to the Cell Broadband Engine, on which a typical streaming application, namely a motion JPEG decoder, is shown to reach an almost linear speed-up. Specifically, this is achieved by tackling two challenges that can severely limit the performance of a process network executing on a multiprocessor:
Granularity of Actors. The granularity of actors in terms of instruction and data memory size or the size of channels and individual tokens influences the performance of a process network. If actors are too fine-grained, the context switching between actors causes a significant execution overhead and communication delays slow down execution. If actors are coarse-grained, the context switching overhead and the influence of communication delays decrease, but it is difficult to equally balance the utilization of resources. Even worse, actors might not fit any more into the memory of some processors, limiting the possibilities for mapping a process network onto a multiprocessor. Facing these trade-offs, one can argue that software developers should rather specify fine-grained actors because coarse-grained actors severely constrain the design space. The challenge is then to keep the overhead for the orchestrated execution of multiple actors at a minimum.

Communication Overhead. The explicit communication between actors via message-passing can considerably limit the speed-up achievable by parallel execution. In particular, when actors are bound to the same processor, message-passing can become inefficient because actors could directly share memory instead of exchanging messages. The challenge is to keep also this communication overhead at a minimum.

On a first view, a run-time environment for executing multiple actors on a single processor appears to come at considerable cost in terms of development effort, run-time overhead, and code size. This belief is also echoed in the following two statements, for instance, that are taken from recent publications about tools for multiprocessor software development based on process networks: “Although multitasking for the SPEs [synergistic processing element on the Cell Broadband Engine] is also possible, in practice it is inefficient as the context switching is very expensive.” [NMSD09] and “... no solution has yet been found to perform an efficient context switch on a DSP that contains 256 40-bit registers and that can potentially execute 8 instructions per cycle.” [CP09].

These conjectures disregard that providing quasi-parallelism to execute actors in a process network is less demanding than providing quasi-parallelism in a fully preemptive event- or time-triggered operating system. Rather, one can employ so-called protothreads [DSVA06] to implement the quasi-parallelism required for executing process networks. Protothreads were originally proposed by Dunkels et al. for programming small embedded systems, such as wireless sensor nodes. Interestingly, the single cores in a multiprocessor are similarly constrained with respect to memory and performance, making protothreads a suitable candidate for providing the required quasi-parallelism.
Furthermore, one can employ so-called windowed FIFOs \cite{HGT07} instead of standard FIFOs for efficient communication. Contrary to standard FIFOs where data need to be explicitly copied into the FIFO buffer, windowed FIFOs permit actors to directly access a portion (window) of the FIFO buffer, thereby avoiding memory copy operations.

The remainder of this chapter is structured as follows: first, the problems associated with the design of a multiprocessor run-time environment for process networks are described. A generic run-time environment for tackling these problems is proposed in Section 3.2. Then, an implementation of the proposed run-time environment for the Cell Broadband Engine is described based on which Section 3.4 presents experimental results. A description of related work and a summary conclude the chapter.

3.1 Problem Statement

In this chapter, the efficient execution of process networks on multiprocessors is considered where “efficient” refers to speed-up. Efficiency in terms of latency, power, or real-time behavior is not considered. The goal is to maximize the speed-up, that is, to reach a speed-up that equals (or even surpasses) the number of processors in a multiprocessor. As applications are specified in a platform-independent manner, this requires a run-time environment that provides the following services:

- mechanism to run multiple actors in a quasi-parallel manner on individual processors,
- FIFO communication between actors bound to a single processor, and
- FIFO communication between actors bound to different processors.

Preferably, this run-time environment should also be scalable and portable, thereby facilitating its implementation on different multiprocessors.

The main challenges faced when devising a suitable run-time environment are related to the characteristics of multiprocessors and the programming model:

**Small Memories.** The data and instruction memories of individual processors on a multiprocessor are usually rather small. The mAgicV DSP of the Atmel Dtiopsis 940, for instance, has a data memory of
80 kBytes and an instruction memory of 128 kBytes. Similarly, each SPE of the Cell Broadband Engine has a local memory of only 256 kBytes for both data and instructions. This does not only limit the space for implementing a run-time environment but possibly also turns memory fragmentation into a serious problem.

**Large Thread Contexts.** The context of an execution thread, that is, the content of general- and special-purpose registers as well as the thread control block, can be rather large on RISC processors and DSPs due to their large register files. The size of the mAgicV DSP register file on the Atmel Dipsis 940, for instance, is 1280 bytes, and the size of the SPE register file on the Cell Broadband Engine is 4096 bytes. This corresponds to roughly 1.6\% of the available local data memory in both cases. Therefore, performing a context switch on these processors is relatively expensive in terms of execution cycles and memory requirements.

**Avoidable Memory Copies.** A standard FIFO implementation that is accessed using `read()` and `write()` routines, as introduced in Section 2.3, causes a significant execution overhead: explicitly copying data to and from the FIFO buffer requires processor cycles and memory space. Instead, if actors could share a part of memory, this overhead could be reduced to the synchronization effort between the involved actors. On the other hand, copying data via message-passing is inevitable when the actors cannot directly access the same memory.

### 3.2 Lightweight Run-Time Environment

This section introduces a multiprocessor run-time environment for process networks that overcomes the challenges listed in the previous section, wherein the basic approach is to use so-called protothreads and windowed FIFOs. The presented approach is motivated by three essential properties of process networks:

**Cooperative Actors.** The actors of a process network are designed to cooperate. Therefore, a run-time environment for running multiple actors of a process network on a single processor does not necessarily require all the mechanisms usually found in a full-fledged multi-tasking operating system, such as full preemption, memory protection, or different scheduling policies. Rather, preemption is only required when actors are blocked due to empty input channels or full output channels, and it is even desirable to share memory between actors to reduce the communication overhead.
Untimed Model. The process network model is an untimed model of computation, which means that a process network can be executed in a completely data-driven manner. Each processor in a system can thus execute autonomously as long as data are available in the input FIFO channels and buffer space is available in the output FIFO channels of the actors bound to that processor. The run-time environments on different processors can thus execute independently of each other.

Separation of Computation from Communication. While the above mentioned aspects are the basis for the efficient execution of (a part of) a process network on a single processor, the key for decent multiprocessor performance is to overlap communication between processors with computations performed on these processors. In a process network, this is facilitated by the explicit separation of computation from communication. In an implementation, direct memory access (DMA) engines can be used for parallelizing computation and communication on most multiprocessors.

3.2.1 Approach

By leveraging the properties mentioned above, a run-time environment for process networks can be implemented using protothreads [DSVA06] and windowed FIFOs [HT07]. Such a run-time environment is efficient and lightweight: first, by keeping the overhead for cooperative execution of multiple actors as well as for communication low, speed-up is only minimally affected by the run-time environment. Second, the overhead is small in terms of instruction and data memory.

This is mainly caused by the use of protothreads to implement actors that are cooperatively executing. Basically, a protothread is an ordinary function with one exception, namely that the execution of a protothread can be resumed at statements that precede a `return` statement. One possible implementation of protothreads proposed in the original paper [DSVA06] is based on a special (but ANSI-C compliant) use of the C language’s `switch` statement, which is referred to as Duff’s device in the community.

Using Duff’s device, protothreads can be implemented by a set of macros that are inserted at the beginning and end of an ordinary function, and a macro that implements preemption points. These macros are shown in Listing 1 and can be used to implement an actor with blocking statements, as shown in Listing 2. A complete example is shown in Appendix A.2.
Listing 1: Macros for implementing protothreads, as proposed in [DSVA06].

```c
struct pt { unsigned short lc; }
#define PT_BEGIN(pt) switch(pt->lc) {
#define PT_WAIT_UNTIL(pt, cond) case 0:
#define PT_END(pt) } pt->lc = 0; return 1
```

Listing 2: An actor implemented as a protothread. Using a non-blocking \texttt{READ()} primitive in combination with \texttt{PT_WAIT\_UNTIL()}, a blocking \texttt{READ()} primitive is implemented.

```c
int protothread(struct pt *pt) {
PT_BEGIN(pt); switch(pt->lc) {
    case 0: ;
    case 4: if (!fifo->read(…)) return 0;
}
PT_END(pt);
pt->lc = 0; return 1;
}
```

The left-hand side of Listing 2 shows a protothread that contains a \texttt{READ()} statement in line 4, which should exhibit blocking behavior, that is, if the channel does not contain sufficient tokens, the execution of the protothread should be discontinued and resumed at the \texttt{READ()} statement upon the next invocation. The statement \texttt{fifo->read()} itself is non-blocking in the sense that it returns false if the channel does not contain enough tokens. Otherwise, it returns true and actually carries out the \texttt{READ()} operation, that is, moves tokens from the FIFO buffer to a local buffer of the actor.

The right-hand side of Listing 2 shows the corresponding code after running the C pre-processor, implementing the desired behavior. When \texttt{fifo->read()} is false, the execution of the protothread will be discontinued and resumed at the case branch immediately before the \texttt{fifo->read()} statement at the next invocation. When \texttt{fifo->read()} is true, however, the macros effectively have no influence on the execution.

Consequently, voluntary preemption and blocking statements can be implemented in this manner, enabling the cooperative execution of multiple actors. Using protothreads thus solves two problems: first, processor context switches can be avoided because invoking a protothread amounts to a simple function call. Second, the memory overhead is very low because the implementation consists of a set of
macros rather than a run-time library. Finally, note that if actors were not cooperative or purely data-driven execution was infeasible, using protothreads would not be possible.

As mentioned before, it is proposed to use windowed FIFO channels \cite{HGT07} for communication instead of standard FIFO channels. By providing actors direct access to the FIFO buffer, data copies can be avoided, as has already been observed in the context of process networks \cite{RvE02,NKG02,vdWK04}. An appropriate API will be introduced in Section 3.2.3.

![Figure 6: Structure of multiprocessor run-time environment for process networks.](image)

The structure of a multiprocessor run-time environment based on protothreads and windowed FIFOs is shown in Fig. 6. Each processor hosts a run-time environment that is similar to the functional simulation on a uniprocessor, as shown in Fig. 5. These run-time environments allow to execute multiple actors on each processor and provide a local windowed FIFO implementation for communication. Additionally a FIFO implementation for the communication between actors on different processors is provided. Being the only platform-dependent component, this implementation usually relies on basic services of an underlying operating system or hardware abstraction layer, such as synchronization primitives, memory mapping, or access to the DMA engines. In the following, further details of the run-time environment are discussed.

### 3.2.2 Protothreads

When running multiple actors on a single processor, it should be possible to switch between actors when blocking occurs. In the following, the term...
thread is used to denote the mechanism that provides the required quasi-parallelism on a single processor. Table 2 shows a comparison of three basic mechanisms to implement threads, namely kernel-space threads, user-space threads, and stack-less user-space threads (protothreads).

Table 2: Comparison of different thread implementations.

<table>
<thead>
<tr>
<th>THREAD SWITCHING MECHANISM</th>
<th>KERNEL-SPACE</th>
<th>USER-SPACE</th>
<th>STACK-LESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>context switch, restoration of thread table</td>
<td>processor context switch, restoration of thread table</td>
<td>stack unwinding</td>
<td></td>
</tr>
<tr>
<td>SCHEDULING</td>
<td>time- or event-driven</td>
<td>event-driven</td>
<td>event-driven</td>
</tr>
<tr>
<td>PREEMPTION</td>
<td>preemption at any point by scheduler</td>
<td>voluntary preemption by threads</td>
<td>voluntary preemption by threads</td>
</tr>
<tr>
<td>STACK</td>
<td>one per thread</td>
<td>one per thread</td>
<td>shared among all threads</td>
</tr>
<tr>
<td>EXAMPLE LIBRARY</td>
<td>POSIX threads (pthreads)&lt;sup&gt;1&lt;/sup&gt;</td>
<td>YAPI&lt;sup&gt;2&lt;/sup&gt;, SystemC&lt;sup&gt;3&lt;/sup&gt;</td>
<td>protothreads&lt;sup&gt;4&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

When comparing the properties of these thread implementations with the requirements, it turns out that protothreads are a good choice for implementing the quasi-parallelism required for process networks: first, a context switch in protothreads basically amounts to returning from one function and calling another one. This takes considerably less time than context switches for kernel- or user-space threads. Second, all threads in a protothreads environment share a single stack. This is an advantage in memory-constrained systems because it avoids memory fragmentation. Third, protothreads are basically implemented using a small set of macros, as shown above. This means that there is also little overhead in terms of code size in contrast to kernel- or user-space threads, which require a separate run-time library for execution. Fourth, protothreads are architecture-independent, requiring only an ANSI-C compliant compiler. Clearly, porting a kernel- or user-space thread library to a platform is
more involved, usually requiring the use of inline assembler for accessing processor registers to implement the thread context switch.

3.2.3 Windowed FIFO Communication

A windowed FIFO permits direct access to a window of the channel buffer, which can, once acquired, be randomly accessed. While using windowed FIFO channels preserves the process network semantics, as has been formally proven in [HGT07], it avoids the costly copying of data required in a standard FIFO implementation because of the possibility to directly access the channel buffer. In [vdWdKH+04], for instance, the execution time of an MP3 decoder could be reduced by approximately 30% by using windowed FIFOs instead of standard FIFOs.

Therefore, it is proposed to use windowed FIFOs instead of standard FIFOs when efficiency is a concern. Contrary to read() and write() for standard FIFOs, there is no established naming convention for windowed FIFO access routines. Hence, the following API is proposed:

• capture() acquires a read window.
• consume() abolishes a read window from a FIFO.
• reserve() acquires a write window.
• release() commits a write window to a FIFO.

capture() and reserve() are blocking and return pointers to a contiguous piece of memory in the FIFO buffer, whereas consume() and release() are non-blocking routines.

**Windowed FIFO Communication on Single Processor.** Using windowed FIFOs for communication instead of standard FIFOs is particularly advantageous for actors that share the same memory because unnecessary copying of data can be completely avoided by directly accessing the FIFO channel buffer. In distributed memory architectures, usually all the actors executing on a single processor share the same local memory and can thus profit from this implementation. In shared memory architectures, this implementation can even be used for actors executing on different processors.

Fig. 7 shows one possible implementation of the windowed FIFO that consists of a ring-buffer and four pointers to store the state of the windowed FIFO. Data are enqueued at the tail and dequeued from the head whereby each of the four windowed FIFO access routines basically just moves one of the four pointers. Since reserve() and capture() are required to return pointers to a contiguous piece of
memory, windows are not permitted to span the “wrapping-around” of the ring-buffer. This case requires special attention but can be solved by (temporarily) extending the start or the end of the ring-buffer, for instance.

![Diagram of windowed FIFO](image)

**Figure 7:** Implementation of windowed FIFO.

**Windowed FIFO Communication between Processors.** While the implementation of windowed FIFO communication on a single processor is platform-independent, the implementation of the windowed FIFO communication between processors is platform-dependent. In any case, to parallelize computation and communication, an implementation will utilize DMA controllers, if available. Apart from that, there are many possibilities for implementing the windowed FIFO. The FIFO buffer can be implemented in the memory of the sender, receiver, or both. Similarly, either the sender or the receiver could issue the DMA request. Finally, also a third processor could be used for the buffering and coordination required in windowed FIFO communication. In Section 3.3, a concrete windowed FIFO implementation for the Cell Broadband Engine is discussed.

### 3.2.4 Limitations

A run-time environment based on protothreads and windowed FIFOs has some limitations and disadvantages as compared to a run-time environment based on kernel- or user-space threads and standard FIFOs with respect to the following aspects:
Cooperative Scheduling. A run-time environment based on protothreads can only implement cooperative scheduling where actors voluntarily cede time to each other. This reduces the applicability of the run-time environment for (hard) real-time systems because these systems usually require fully preemptive scheduling policies, such as earliest deadline first or time-triggered policies. Further, when an actor does not give up the processor — usually, actors yield execution when blocking on a FIFO access or after completion of a firing — the entire system hangs up.

Code Organization. Even though the implementation of actors using protothreads as shown in Listing 2 is subject to software synthesis and thus transparent to the programmer, using it incurs some restrictions on the code organization: one restriction is that using a switch statement is not permitted in the fire() routine of an actor. A second restriction is that the content of automatic variables, that is, variables with function-local scope that are automatically allocated on the stack, are not preserved across blocking statements. Therefore, these variables need to be explicitly saved in the local state of an actor. Finally, FIFO access routines must be directly contained in the fire() routine because blocking statements cannot be used in (nested) functions called from a protothread.

These restrictions can be easily checked by static code analysis and compared to the advantages of protothreads, the loss of flexibility for organizing the application code appears to be a small penalty.

Missing Encapsulation. A windowed FIFO provides direct access to the channel buffer, which violates the encapsulation of the implementation. Similarly, protothreads do not encapsulate the state of actors or provide any memory protection. In both cases, this leads to an increased vulnerability to programming errors. Functional simulation in a uniprocessor run-time environment based on protothreads and windowed FIFOs can be used to (partially) expose potential problems at an early software development stage, however.

3.3 Efficient Execution of Process Networks on the Cell Broadband Engine

In the previous sections, the basic concepts that allow the efficient execution of process networks on multiprocessors have been described. To demonstrate the viability of these concepts, a prototype implementation of the proposed run-time environment has been integrated into the DoI for one specific multiprocessor, namely the Sony/Toshiba/IBM
Cell Broadband Engine (Cell BE) [PAB+06]. Note that porting the implementation to other multiprocessors is relatively easy because of the small number of services and the facility to implement protothreads without using assembly code.

### 3.3.1 Sony/Toshiba/IBM Cell Broadband Engine

The Cell BE is a heterogeneous multiprocessor that consists of one 64-bit PowerPC processing element (PPE) and eight synergistic processing elements (SPEs). All processors are clocked at a frequency of 3.2 GHz and are interconnected by a ring bus clocked at 1.6 GHz. A high-level block diagram of the Cell BE is outlined in Fig. 8.

**Figure 8:** Simplified block-diagram of the Cell Broadband Engine.

While the PPE is a PowerPC processor, the SPEs are RISC processors with 128-bit single-instruction multiple-data (SIMD) organization that can process up to four 32-bit integers or single-precision floating point values per clock cycle. Using load and store instructions, an SPE can only access its own local store, a small memory of 256 kBytes. A DMA engine enables the access to the main memory and the communication with other SPEs. The element interconnect bus (EIB) that interconnects all processors and the memory interface controller (MIC) consists of four data rings (two running clockwise and two running counterclockwise). The SPEs are connected to the EIB through memory flow controllers (MFC), which manage DMA requests of SPEs. In addition, the MFCs host small FIFO channels, the so-called mailboxes, which can be used to exchange 32-bit messages between processors.
Regarding the implementation of process networks on the Cell BE, the following restrictions have to be taken into consideration:

- Preemptive scheduling is fully supported on the PPE by using POSIX threads. There is no kernel- or user-space thread library for the SPEs available, however.

- Due to the low bandwidth and high delay of mailbox communication, mailboxes can only be efficiently used for exchanging synchronization messages, but not for data transfers.

- Due to the hardware architecture of the MFC, the source and the destination buffers of a DMA transfer must be “aligned”. A memory block of $N$ bytes is called “aligned” when more than the seven least significant bits (or at least the $\log_2 N$ least significant bits) of its address are zero.

### 3.3.2 Process Network Run-Time Environment for the Cell BE

The run-time environment to execute a process network on the Cell BE basically follows the principles explained in Section 3.2. This section explains several platform-dependent details of a prototype implementation of this run-time environment on the Cell BE. Note that the run-time environment runs on top of a Linux operating system that provides basic services, such as memory management, disk access, and device drivers, and is also used for bootstrapping, starting, and stopping the run-time environment. In addition, routines provided by the Cell BE software development kit (SDK) are used for the device-level implementation of the communication routines.

**Quasi-Parallelism on Single Processor.** On the PPE as well as on each SPE, multiple actors are executed in a quasi-parallel manner using protothreads. The actors bound to a processor are wrapped in a single execution thread, and these threads are then dispatched by the PPE to the individual SPEs, except for the actors bound to the PPE, which remain (wrapped in a single POSIX thread) on the PPE. After bootstrapping a process network in this manner, an application can execute completely distributed.

**Windowed FIFO Communication on Single Processor.** Windowed FIFOs that connect actors executing on the same processor are implemented according to Fig. 7. In case of the PPE, the channel buffer is implemented on the main memory, whereas for the SPEs, it is implemented on the corresponding local store.
Windowed FIFO Communication between Processors. After bootstrapping a process network, the single processors of the Cell BE only interact via windowed FIFOs. The developed protocol is sketched in Fig. 9, which shows that windowed FIFO communication only involves the sending and receiving actors and processors and, thus, does not require any global coordination. The basic principle underlying the protocol is that data are forwarded as soon as possible from the sender to the receiver. This means that data transfers from the local store of the sender to the local store of the receiver are initiated as soon as the sender has invoked the `release()` routine. The receiver regularly checks whether data are ready and carries out the corresponding transfer.

The protocol works as follows: whenever the sender has released a window, it informs the receiver that new data are available. Afterwards, the sender can continue as long as it is not blocking on a channel. In that case, other actors that are ready to execute could be executed. The actual transfer is carried out by the receiver, which checks whether new data are available on every invocation of `capture()` or whenever an actor has finished its `fire()` routine. If a message about the availability of new data is present, the receiver sets up a corresponding DMA transfer to copy the data from the remote memory into its own local store. Due to the alignment restriction, however, the MFC cannot directly copy data between the involved windowed FIFO buffers. To deal with this issue, an aligned temporary buffer is used at the receiver that serves as the destination for the DMA transfer. If the window of the sender is not aligned, the source address is changed to the largest aligned address that
is smaller than the source address, while increasing the number of bytes to transfer accordingly. After the DMA transfer has been set up, the receiver blocks until the transfer is completed. Similar to the sender side, other actors could be executed meanwhile. Finally, the receiver informs the sender about the completion of the transfer, possibly unblocking a `reserve()` routine invoked earlier at the sender. To complete the transfer, the received data are copied from the (aligned) temporary buffer to the (potentially unaligned) buffer of the windowed FIFO, discarding the bytes that have been transferred due to the alignment restriction.

### 3.4 Experimental Results

This section presents several experiments demonstrating the performance of dataflow process networks running on top of the developed run-time environment on the Cell BE. The section also includes a comparison of the performance of different thread implementations. For this purpose, uniprocessor run-time environments based on protothreads, pthreads, SystemC, and the YAPI library have been implemented. The goal is to demonstrate that the proposed run-time environment is, indeed, efficient and lightweight by highlighting the following aspects:

- low runtime overhead for a uniprocessor implementation,
- high bandwidth and speed-up close to the theoretic limit in the case of a multiprocessor implementation,
- execution of fine-grained process networks with the efficiency of a coarse-grained implementation, and finally,
- implementation in a small memory footprint.

#### 3.4.1 Uniprocessor Performance

**Experimental Setup.** To assess the uniprocessor performance, synthetic process network applications using the four run-time environments mentioned above are executed under Linux (kernel 2.6.28) running on an Intel Xeon processor clocked at 3.06 GHz. The used compiler is `gcc 4.1.2` and the optimization level `-O2` is used for all experiments. (The reason for running the uniprocessor experiments on an Intel Xeon processor rather than on the PPE or the SPE of a Cell BE is that suitable ports of the SystemC and the YAPI library were not available for the Cell BE.)

To measure the context switching and communication times, two synthetic applications have been developed, referred to as `singlefifo`
and PINGPONG. The SINGLEFIFO application (Listing 3) is used to measure FIFO access times, its basic functionality being to write/read to/from a FIFO channel that forms a self-loop with the only actor. As only a single actor is executed, no context switches occur. On the other hand, the PINGPONG application (Listing 4) has been implemented to measure context switching times. When running PINGPONG, the run-time environment is forced to switch between the two actors after each invocation of \texttt{fire}() because the actors communicate via FIFO channels with a capacity that equals the size of the communicated tokens. The context switching time is then estimated by subtracting the FIFO access times from the total execution time. For comparison, both applications were implemented using standard and windowed FIFOs.

\textbf{Listing 3:} SINGLEFIFO application for measuring FIFO access times.
\begin{verbatim}
procedure process\_fire(ProcessData *p)
    for (i = 0 to ITERATIONS - 1)
        p->fifo->write(*buf, size);
        p->fifo->read(*buf, size);
    end for
    for (i = 0 to ITERATIONS - 1)
        p->wfifo->reserve(*buf, size);
        p->wfifo->release();
        p->wfifo->capture(*buf, size);
        p->wfifo->consume();
    end for
end procedure
\end{verbatim}

\textbf{Listing 4:} PINGPONG application for measuring context switching times.
\begin{verbatim}
procedure process\_1\_fire(ProcessData *p)
    for (i = 0 to ITERATIONS - 1)
        p->fifo1->write(*buf, size);
        p->fifo2->read(*buf, size);
    end for
end procedure

procedure process\_2\_fire(ProcessData *p)
    for (i = 0 to ITERATIONS - 1)
        p->fifo1->read(*buf, size);
        p->fifo2->write(*buf, size);
    end for
end procedure
\end{verbatim}

\textbf{Context Switching Time.} In Fig. 10, the context switching time and the (windowed) FIFO access time are compared for the four run-time environments (protothread, pthread, SystemC, and YAPI). As expected,
the protothread implementation introduces the smallest context switching overhead. Compared to a user-space thread implementation (YAPI or SystemC), a protothread context switch is about 8 to 18 times faster. Compared to pthreads, a protothread context switch is approximately 200 times faster. Note that the measured context switching times include the execution of the scheduler, as well.

Figure 10: Comparison of execution times for a context switch and a (windowed) FIFO access in different thread implementations. The durations of a context switch and a windowed FIFO access in the protothreads implementation amount to roughly 300 and 150 clock cycles, respectively, thus enabling the efficient execution of rather fine-grained process networks.

(Windowed) FIFO Communication. To measure and compare the execution time for accesses to windowed and standard FIFOs, the singlefifo application was used. The results are shown in Fig. 10, where FIFO access means a single invocation of a \texttt{read()} or \texttt{write()} routine and windowed FIFO access either means a \texttt{reserve()/release()} pair or a \texttt{capture()/consume()} pair, respectively. (For both FIFO types, the read and write routines take approximately the same time.) While the standard and windowed FIFO access times are similar for small accesses, windowed FIFOs are considerably more efficient for large accesses: in case of the windowed FIFO, basically only the pointers shown in Fig. 7 have to be updated. The duration of the involved operations is thus independent from the actual number of transmitted bytes. In the case
of the standard FIFO, however, the data to transmit actually have to be copied to and from the FIFO buffer, increasing the access time as the number of transmitted bytes increases.

**Code Size.** Table 3 shows that in addition to high performance, a run-time environment based on protothreads introduces only a small overhead in terms of code size.

**Table 3:** Size of the binary of the **PINGPONG** application linked against different process network run-time environments.

<table>
<thead>
<tr>
<th></th>
<th>PROTO-THREADS</th>
<th>YAPI</th>
<th>SYSTEMC</th>
<th>PTHREADS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SIZE [kBytes]</strong></td>
<td>42</td>
<td>1443</td>
<td>1343</td>
<td>461</td>
</tr>
<tr>
<td><strong>SIZE AFTER STRIPPING [kBytes]</strong></td>
<td>9</td>
<td>113</td>
<td>221</td>
<td>32</td>
</tr>
</tbody>
</table>

### 3.4.2 Multiprocessor Performance

In the first part of this section, it was shown that a run-time environment based on protothreads combined with windowed FIFO communication enables the efficient execution of process networks on uniprocessors. Now, the performance on a multiprocessor is examined.

**Experimental Setup.** A Sony PlayStation 3 running Yellow Dog Linux 6.1 (kernel 2.6.23) is used for the experiments. The Sony PlayStation 3 contains a single Cell BE whereby the PPE and six SPEs are available for user applications. The gcc compilers `ppu-g++ 4.1.1` and `spu-g++ 4.1.1` are used with the optimization level `-O5`.

**Context Switching and FIFO Communication on the Cell BE.** Using the same applications as in the uniprocessor evaluation, the context switching time and the standard and windowed FIFO access times are measured for the PPE and an SPE. The results are plotted in Fig. [1], which shows the same trends that have already been observed on the Intel Xeon in the uniprocessor evaluation.

**Data Transfer Rate.** To measure the peak data transfer rate between SPEs, two experiments are performed: first, a chain of six actors is mapped onto the six SPEs (one actor per SPE). Second, a chain of 12 actors is
mapped onto the six SPEs (two actors per SPE). To connect the actors in the chains, windowed FIFO channels with a size of 16384 bytes are used, which corresponds to the maximum size of a single DMA transfer on the Cell BE.

**Figure 11:** Comparison of the execution times of a context switch and various FIFO accesses on the Cell BE.

Fig. 12 shows the aggregate inter-SPE data transfer rates for the two chains whereby the number of bytes transmitted in a single windowed FIFO access is varied between 512 bytes and 16384 bytes. Considering the lower end of the spectrum, one can see that the data rate is quite low for small token sizes because the communication delays are dominant. At the upper end of the spectrum, the observed peak data rates are 9.8 GBytes/s when one actor is executed on each SPE, and 10.9 GBytes/s when two actors are executed on each SPE. In the latter case, the data rate is higher because the data transfers initiated by the two actors on each SPE can be partially overlapped.

Even higher data rates have been reported in [Pak08], where an aggregate data rate of approximately 17.3 GBytes/s has been achieved for an MPI implementation on the Cell BE in a similar setup. That implementation assumes aligned source and destination buffers, thereby saving the memory copy operations required in the proposed implementation.
Speed-Up due to Parallelization. To evaluate the speed-up of applications that can be achieved using the proposed framework, a motion JPEG (MJPEG) decoder is used. MJPEG is a video codec in which each video frame is separately compressed as a JPEG image [Wal92].

The process network and mapping used for this evaluation are depicted in Fig. 13, whereby two versions of the process network with different actor granularities are considered: in the coarse-grained implementation, a complete frame is decoded in each of the decode frame actors. In the fine-grained implementation, the decode frame actor
splits a frame into segments of 40 macroblocks that are then forwarded to individual actors performing inverse quantization, zigzag scan, and inverse discrete cosine transform. In both cases, the split stream actor reads the video stream from a file and dispatches single video frames to the subsequent actors. Merge stream collects the decoded frames and displays them. Each decode frame actor (with its associated “child” actors in the case of the fine-grained implementation) is mapped to an SPE. Split stream and merge stream are mapped to the PPE.

In Fig. 14, the time for decoding 3100 frames using the MJPEG algorithm is compared for implementations on a different number of SPEs. Obviously, the peak performance can be achieved when a decode frame actor is mapped to each of the six SPEs. In that case, a grayscale video of $320 \times 240$ pixels can be decoded with a frame rate of $942$ frames/s. Mapping all actors to the PPE leads to a frame rate of $136$ frames/s. Hence, a speed-up of almost seven can be achieved when using the PowerPC and all six SPEs. The plot also shows that by leveraging protothreads and communication via windowed FIFOs, the overhead of the fine-grained version as compared to the coarse-grained version is only about $5\%$.

**Figure 14:** Execution time and speed-up of the MJPEG application for different numbers of SPEs. The scale for the speed-up of the coarse-grained implementation is shown on the right-side y-axis.
3.5 Related Work

There are basically two options to accomplish the efficient execution of applications on multiprocessors with a distributed memory hierarchy without resorting to device-level programming, namely using a high-level API or a model-driven development approach.

When using a high-level API for parallel programming, such as OpenMP [OMP], MPI [MPI], MCAPI [MCA], Multiflex [PPL+06], or TTL [vdWK04], a software developer writes code that can be automatically refined to a given multiprocessor and run-time environment. Due to the lack of an underlying model of computation, however, design activities such as analytic performance analysis are difficult to automate when using one of these APIs. Furthermore, the mapping is implicitly contained in the application code, making it difficult to quickly explore different mappings or port the application to a different multiprocessor.

These disadvantages can be overcome by using a model-driven development approach. In the context of the design flows related to the Dol, see Section 2.5, run-time environments for functional simulation on uniprocessors and efficient execution on multiprocessors have been developed: run-time environments for functional simulation are mostly based on POSIX threads, such as in C-Heap, Daedalus, and Shim. Note that the Shim run-time environment was also employed on an Intel quad-core processor to evaluate the potential of a design flow based on process networks for programming symmetric multiprocessors [EVT08].

An exception to the usual approach of using a kernel-space library is the open-source library YAPI [dKES00] developed at the Philips Research Laboratories, which is based on a custom user-space thread implementation. In addition, YAPI features a deadlock detector that allows to automatically increase the size of the FIFO channels during run-time to prevent artificial deadlocks.

With respect to multiprocessor run-time environments, different approaches have been proposed:

- Some design flows use custom run-time environments that do not implement a multi-threading mechanism on all processors such that only a single actor can be mapped onto these processors. Examples of such environments are two process network run-time environments [VE09, NMSD09] for the Cell BE, for instance, where multi-threading on single SPEs is not supported.

- Some design flows are based on existing operating systems, such as Koski which employs eCos [ECO].
A third group of design flows employs static scheduling of multiple actors on a single processor, which is only possible when restricting the model of computation to a subclass of dataflow process networks. An example is the Cell BE run-time environment \cite{ZLRA08} of StreamIt, which is based on synchronous dataflow.

In the case of the Cell BE, run-time environments have been developed in the context of Peace \cite{KLPH08}, Daedalus \cite{NMSD09}, Shim \cite{VE09}, and StreamIt \cite{ZLRA08}. They have in common that multiprocessing on the individual SPEs is very limited. Either only a single actor is executed on each SPE or, in the case of multiple actors, the scheduling is static and actors are non-preemptive.

In this chapter, it has been shown that a run-time environment based on protothreads and windowed FIFOs is advantageous compared to the above mentioned approaches. It has a smaller run-time overhead and a smaller memory footprint without sacrificing portability or the possibility to cooperatively execute multiple actors.

3.6 Summary

Currently, software development for heterogeneous multiprocessors with distributed memory architectures usually takes place at a rather low abstraction level because software standards or design patterns that would allow moving to higher abstraction levels have not been widely adopted thus far. Software developers therefore resort to device-level programming or use a high-level API to implement applications. The source code developed in this manner, however, is usually difficult to reuse and maintain because the mapping of the application onto the architecture is implicitly contained. Hence, mapping an application in a different manner, porting it to a different platform, or analyzing its performance usually requires manual efforts.

Model-driven development is an approach to overcome this problem, wherein the idea is to automatically create a platform-dependent implementation from a high-level specification. In this chapter, it has been shown that this is actually possible for streaming applications modeled as dataflow process networks. To achieve this goal, a suitable run-time environment has been proposed in which protothreads are used to implement quasi-parallelism on the single processors of a multiprocessor and windowed FIFOs to implement the communication between actors. This keeps the runtime-overhead low such that speed-ups close to the theoretical bound can be achieved. The run-time environment can be easily ported to different architectures because only the implementation
of the windowed FIFO communication between different processors is architecture-dependent. The viability of the run-time environment was verified by implementing a prototype for the Cell BE. Without writing architecture-specific code, an MJPEG application executing on top of this run-time environment achieved a speed-up close to seven on seven processors.
Modeling Actors in Modular Performance Analysis

During the design of multiprocessor streaming applications, software developers do not only face questions related to the functional correctness of an application, but also questions related to non-functional properties, such as performance, energy consumption, or code size. Especially timing properties are often particularly interesting because many streaming applications are real-time applications. Therefore, performance analysis, that is, the quantitative analysis of timing properties, plays an instrumental role in the design of multiprocessor streaming applications.

In this chapter, hard real-time systems are considered, that is, systems where it is imperative that timing constraints are satisfied because not meeting a constraint is considered a system failure. Usually, analytic best-case/worst-case methods are employed for performance analysis in this context. Extending classical uniprocessor scheduling theory, different approaches have been presented in the literature, including compositional approaches, such as modular performance analysis (Mpa) [CKT03] and symbolic timing analysis for systems (SymTA/S) [HHJ+05], holistic scheduling analysis [HCC01], and analysis based on timed automata [HV06], see [PWT+09] for a survey. These methods share several advantages over other methods regarding the following aspects:

**System Verification.** To check whether timing constraints are met, safe bounds on timing properties need to be determined during the verification of hard real-time systems. Being capable of covering all possible system behaviors, analytic best-case/worst-case methods are
exhaustive and, therefore, well suited for this purpose. Contrary, empirical methods, such as measurement or simulation at different abstraction levels, can usually not be used to determine hard bounds because they only cover a fraction of all possible system behaviors. Likewise, average-case or probabilistic methods are not suited, either.

**Analysis Time.** Due to the employed abstractions, analytic methods can be faster than other performance analysis methods. Specifically, this applies to compositional approaches that are computationally efficient because the analysis complexity grows only linearly with the number of components. Holistic and stateful models often suffer from an exponential growth in complexity, inhibiting their application to large systems.

**Design Space Exploration and System Optimization.** To optimize the performance of multiprocessor streaming applications, designers have to deal with the binding and scheduling of actors or the allocation and placement of FIFO buffers, for instance. In this context, analytic performance analysis can be useful because it is based on an abstract model to analyze a system. Frequently, certain properties of that model can be leveraged in dedicated optimization techniques to accelerate the optimization process. Consequently, these techniques usually outperform corresponding black-box optimization techniques.

On the other hand, one can identify the following drawbacks of analytic best-case/worst-case methods:

**Limited Modeling Scope.** Analytic best-case/worst-case methods are associated with a system model that covers event stream models, the model of computation, and resource sharing policies, among others. A system that does not closely fit the model underlying a specific method either cannot be analyzed at all or only with a considerable loss of accuracy. Hence, there is great interest in extending the modeling scope of individual methods and in combining different methods to encompass a wide range of systems.

**Large Modeling Effort.** To be able to carry out analytic performance analysis, expert knowledge about the system itself but also about the specific analysis method is required. Due to the increasing size of systems and the steadily increasing scope of analytic methods, gathering this knowledge and applying it to create a performance analysis model gets more and more difficult.
In this chapter, the problem of limited modeling scope is addressed, the second problem is addressed in the next chapter. Considering streaming applications and dataflow process networks, one can observe that the scope of analytic performance analysis methods is frequently rather limited with respect to the modeling of individual actors. In compositional methods like Mπ and SymTA/S, for instance, the basic model assumes actors with a single input and a single output event stream. Extensions to model actors with several input streams are, so far, limited to basic activation schemes like OR- and AND-activation \cite{Wan06, JRE05}. In real systems, however, actors might exhibit more complex activation schemes and consist of sub-actors being activated by tokens occurring in a subset of input streams.

In the following, this shortcoming of compositional methods is partly removed by introducing a method to analyze actors with complex activation schemes. For this purpose, first a generic actor model is introduced that captures a wide range of activation schemes. Second, a best-case/worst-case analysis method for actors complying with this model is proposed in the framework of Mπ. In this context, a novel method to analyze actors belonging to a non-preemptive fixed priority scheduling domain is presented. Furthermore, previous results for OR- and AND-activated actors with two inputs reported by Wandeler \cite{Wan06} are generalized to multiple inputs.

This chapter is structured as follows: first, the actor model considered in this chapter is introduced. Afterwards, the general approach to analyze actors that are structured according to this model is described. In Section 4.3, Mπ and the notations used are introduced, and in Section 4.4, the analysis of actors is presented in detail. Subsequently, the method is applied in several application scenarios and compared to other methods. The chapter concludes with a description of related work and a summary.

4.1 Actor Model

In this chapter, the best-case/worst-case analysis of actors with complex activations schemes is considered. The goal is to cover two design patterns of practical relevance that are exemplified in Listing 5.

Multiple Inputs. In realistic systems, the flow of data between actors is usually not limited to one-to-one connections. Rather, the activation of an actor frequently depends on tokens arriving on input streams from multiple actors. In this case, multiple event streams need to be related in a way that correctly describes the semantics of the activation encountered in the real system.
Multiple Sub-Actors. Sometimes, it is desirable to integrate multiple actors into one actor. One approach to do so is a data-driven execution scheme where the activation conditions of the individual sub-actors are sequentially checked in a non-blocking fashion and a sub-actor is fired whenever its activation condition is fulfilled. This scheme could also be implemented by the run-time environment proposed in the previous chapter, for instance, given that individual sub-actors follow the read-execute-write pattern in Listing 5.

Listing 5: Example of an actor with multiple sub-actors exhibiting complex activation schemes. The actor is assumed to be repeatedly called by a scheduler. The non-blocking test(stream, N) function returns true if N tokens are available on the specified stream, otherwise false. The parameter i indicates that the indexed parameters may vary between activations.

```plaintext
if test(inputA, A_in(i))
  remove A_in(i) tokens from inputA;
  execute code a(i);
  send A_out(i) tokens to outputA;
else if test(inputB1, B1_in(i)) or test(inputB2, B2_in(i))
  remove B1_in(i) tokens arrived at inputB1 or
  B2_in(i) tokens arrived at inputB2,
  depending on which are available;
  execute code b(i);
  send BA_out(i) tokens to outputA;
  send BB_out(i) tokens to outputB;
else if test(inputC1, C1_in(i)) and test(inputC2, C2_in(i))
  remove C1_in(i) tokens from inputC1 and
  C2_in(i) tokens from inputC2;
  execute code c(i);
  send C_out(i) tokens to outputC;
end if
```

More specifically, the following patterns can be identified in Listing 5:

- A single actor is split up into sub-actors where the input streams of the sub-actors are pairwise disjoint. Several sub-actors might emit data to the same output stream, however.

- The activation conditions of sub-actors are checked in a non-blocking fashion, following a pre-defined sequential order. When an activation condition is fulfilled, the corresponding sub-actor is executed, and afterwards the checking of activation conditions starts from the beginning.

- The activation conditions of sub-actors are defined as conditions on the availability of tokens. In analogy to Boolean logic, activation conditions are expressed as Boolean combinations of OR and AND where the presence of a token corresponds to a logic 1 and its absence to a logic 0.
4.2 Approach

Specifically, OR-activation means that whenever an event is available on any input stream, the actor can fire. AND-activation, sometimes also referred to as ALL-activation, means that whenever an event is available on all input streams, the actor can fire. By combining OR- and AND-activation, arbitrary Boolean activation patterns excluding negations, that is, the non-existence of events, can be described.

- Variability of the actor behavior across different firings is taken into consideration in several ways: first, the number of tokens required to activate a sub-actor might differ for different activations. Second, the actual code that is executed might vary from firing to firing and, consequently, the execution time of an actor. Third, also the number of produced tokens might vary.

The specific problem considered in this chapter is to derive best-case and worst-case bounds on the timing behavior of components following the structure in Listing 5. In the context of Mα, this requires deriving the input–output relations of arrival and service curves of a component modeling this actor. Based on these relationships, specific quantities can be computed, such as upper bounds on the backlog occurring on the individual input streams or upper bounds on the delay for processing tokens of the individual input streams.

4.2 Approach

To analyze actors with complex activation schemes, a compositional approach is taken. Basically, the analysis model for an actor is split up into multiple components for which individual input–output (transfer) functions are individually derived. The analysis of an entire actor is then performed by connecting the components, as shown in Fig. 15.

The basic component of the analysis is the so-called greedy processing component (GPC) which models the processing of a single event stream under resource constraints [CKT03]. Variable execution time of actors is taken into consideration by leveraging so-called workload curves [MKT04] within a GPC. For the analysis of sub-actors processing multiple input streams, the abstract OR and abstract AND components proposed in [Wan06] are used. These components allow determining the activation pattern resulting from Boolean activation conditions. To deal with the varying number of consumed and produced tokens, so-called consumption curves $\kappa$ and production curves $\pi$ [MZCW04] are used that give upper and lower bounds on the number of consumed and produced tokens per activation. Summarizing, greedy processing
components, abstract OR and AND components, and consumption and production curves are used to model individual sub-actors.

To analyze an actor with multiple sub-actors, one needs to adequately model the way the computation resources available to the actor are shared by its sub-actors. To this end, the structure in Listing 5 implies that computation time is distributed as in a non-preemptive fixed priority scheduler: a sub-actor whose activation condition is checked first will be executed as long as data are available. Only if this is not the case any more, will the activation condition of the second sub-actor be checked and executed, if possible, and so forth. Furthermore, as soon as a sub-actor has started a firing, it will complete this firing. As in a non-preemptive scheduler, it cannot be interrupted by any of the other sub-actors (of “higher priority”) even if their activation condition would get fulfilled in the mean time. Therefore, it is possible to model the actor as a non-preemptive fixed priority scheduling domain. This scheduling domain can then be seamlessly embedded into the Mpa model of an entire system due to the compositional approach of Mpa and, specifically, the possibility of analyzing hierarchical scheduling domains.

4.3 Modular Performance Analysis

In the domain of real-time streaming applications, powerful abstractions have been developed to model and analyze distributed systems. The framework used in this thesis is modular performance analysis (Mpa) [CKT03]. With Mpa, hard upper and lower bounds can be computed for
various performance criteria in a distributed real-time system, such as end-to-end delays, buffer requirements, or resource utilization. Hence, \textsc{Mpa} qualifies for the analysis of hard real-time systems. This clearly distinguishes \textsc{Mpa} from any probabilistic performance analysis method and from empirical methods based on measurements or simulations.

Basically, the analysis of real-time systems requires knowledge about the best-case and worst-case behavior of the system in all operating conditions. In \textsc{Mpa}, a component-based approach is employed to tackle this problem: A system is split up into “small” components with no or little interference that are characterized independently from each other. Afterwards, the interference of components is modeled to globally analyze a system. This takes into account the interaction of software and hardware components with respect to resource sharing, data dependencies, or operating system overhead.

Concretely, the \textsc{Mpa} model of a system, as shown in Fig. [16], is composed of abstract elements modeling the event streams that carry data and trigger actors, the computation and communication of an application, the resources, such as processors and interconnects, and the resource sharing methods. The approach uses real-time calculus (RTC) \cite{CKT03} which itself is based on the theoretical framework of network calculus \cite{Cru91a, Cru91b, LBT01}. In particular, arrival curves $\alpha(\Delta)$, service curves $\beta(\Delta)$, and workload curves $\gamma(e)$ model the timing properties of event streams, the capability of architecture elements, and the execution requirements of event streams, respectively. Abstract components define the semantics of task execution and resource sharing in the system. A short description of these elements is given next. For further details, the reader is referred to \cite{WTVL06}.

**Figure 16:** \textsc{Mpa} model of a system where preemptive fixed priority scheduling is used on all resources (left) and internals of a greedy processing component (right).
Figure 17: Arrival and service curves of a single greedy processing component. The component with an execution time of 3 million cycles is periodically activated with a period of 5 ms ($\alpha$). It is executed on a resource whose availability is modeled by a bounded delay model ($\beta$). The output event stream is still periodic, but has a jitter of 3 time units ($\alpha'$). The remaining resource availability is rather irregular ($\beta'$).

4.3.1 Event Stream Model

Event streams in a system can be described using a cumulative function $R(s, t)$, defined as the number of events seen in the time interval $s \leq \tau < t$. While any $R$ always describes one concrete trace of an event stream, a tuple $\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)]$ of upper and lower arrival curves provides an abstract event stream model that characterizes a whole class of (non-deterministic) event streams. Specifically, $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ provide an upper and a lower bound on the number of events seen on the event stream in any time interval of length $\Delta$.

Definition 1: (Arrival Curve) Let $R(s, t)$ denote the number of events that arrive on an event stream in the time interval $s \leq \tau < t$. Then, the corresponding upper
and lower arrival curves are denoted by $\alpha^u$ and $\alpha^l$, respectively, and satisfy:

$$\alpha^l(t-s) \leq R(s, t) \leq \alpha^u(t-s) \quad \forall s < t \quad s, t \in \mathbb{R}_0^+$$  \hspace{1cm} (4.1)

where $\alpha^u(0) = \alpha^l(0) = 0$.

In this model, the timing information of standard event stream models like periodic, periodic-with-jitter, periodic-with-bursts, sporadic, or any other arrival pattern with non-deterministic timing behavior can be represented by an appropriate choice of $\alpha^u$ and $\alpha^l$ \cite{CKT03}. In Fig. 17 for instance, $\alpha$ represents the arrival curves for a periodic stream whereas $\alpha'$ represents the arrival curves of a periodic stream with jitter. In general, arrival curves are suited to represent the complex timing behaviors of event streams in multiprocessors.

### 4.3.2 Resource Model

In a similar way, the capability of computation or communication resources can be described by a cumulative function $C(s, t)$, defined as the number of available resources, that is, processor or bus cycles, in the time interval $s \leq \tau < t$. To provide an abstract resource model that models a whole set of possible resource behaviors, one can define service curves as follows.

**Definition 2: (Service Curve)** Let $C(s, t)$ denote the number of clock cycles that a resource can provide in the time interval $s \leq \tau < t$. Then, the corresponding upper and lower service curves are denoted as $\beta^u$ and $\beta^l$, respectively, and satisfy:

$$\beta^l(t-s) \leq C(s, t) \leq \beta^u(t-s) \quad \forall s < t \quad s, t \in \mathbb{R}_0^+$$  \hspace{1cm} (4.2)

where $\beta^u(0) = \beta^l(0) = 0$.

Again, service curves substantially generalize classical resource models, such as the bounded delay or the periodic resource model. In Fig. 17, $\beta$ represents the service curves for a bounded delay resource whereas $\beta'$ represents the irregular service that would be left for an actor with lower priority.

### 4.3.3 Workload Model

To relate arrival and service curves, workload curves are used \cite{MKT04}. The workload that the processing of an event imposes on a computation or communication resource can be described by a cumulative function $W(e)$ defined as the number of clock cycles required to process the first $e$ consecutive events of an event stream.
Chapter 4. Modeling Actors in Modular Performance Analysis

**Definition 3: (Workload Curve)** Let \( W(e) \) denote the number of clock cycles that are required by a software or hardware component to process the first \( e \) consecutive events of an event stream. Then, the corresponding upper and lower workload curves are denoted as \( \gamma^u \) and \( \gamma^l \), respectively, and satisfy:

\[
\gamma^l(t - s) \leq W(t) - W(s) \leq \gamma^u(t - s) \quad \forall s < t, \ t \in \mathbb{N}_0
\]

(4.3)

where \( \gamma^u(0) = \gamma^l(0) = 0 \).

Workload curves generalize the usual characterization of software components by their best-case and worst-case execution time. Specifically, workload curves allow a tighter characterization of the execution time of components because the execution time to process a sequence of events is often smaller than the sum of the worst-case execution times to process one event. Analogously, the execution time to process a sequence of events is often larger than the sum of the best-case execution times.

According to Definitions 1 and 2, arrival curves are event-based whereas service curves are resource-based. Using the workload curve and its pseudo-inverse \([LBT01]\),

\[
(\gamma^u)^{-1}(w) = \sup\{e : \gamma^u(e) \leq w\}, \quad (\gamma^l)^{-1}(w) = \inf\{e : \gamma^l(e) \geq w\},
\]

(4.4)

(4.5) and (4.6) describe how arrival and service curves can be transformed from event-based to resource-based quantities and vice versa, as shown in Fig. 16.

\[
\bar{\alpha}^u(\Delta) = \gamma^u(\alpha^u(\Delta)) \quad \bar{\beta}^u(\Delta) = (\gamma^l)^{-1}(\beta^u(\Delta))
\]

(4.5)

\[
\bar{\alpha}^l(\Delta) = \gamma^l(\alpha^l(\Delta)) \quad \bar{\beta}^l(\Delta) = (\gamma^u)^{-1}(\beta^l(\Delta))
\]

(4.6)

In the simplest case, the workload of an actor is characterized by its best-case and worst-case execution demand (BCED and WCED) measured in clock cycles. In this case, the workload curves and their inverse are defined as follows:

\[
\gamma^u(e) = \text{WCED} \cdot e \ [\text{cycles}] \quad (\gamma^u)^{-1}(x) = \lfloor x/\text{WCED} \rfloor \ [\text{events}]
\]

(4.7)

\[
\gamma^l(e) = \text{BCED} \cdot e \ [\text{cycles}] \quad (\gamma^l)^{-1}(x) = \lceil x/\text{BCED} \rceil \ [\text{events}]
\]

(4.8)

### 4.3.4 Actor Model

In MPA, the processing of event streams by software or hardware components is modeled by greedy processing components (GPC). The semantics of a GPC can be described as follows: an incoming event stream, represented by an upper and a lower arrival curve, flows into the
FIFO input buffer of the GPC. Each event triggers an execution which is restricted by the availability of resources, represented by an upper and a lower service curve. The output event stream can again be represented by an upper and a lower arrival curve, whereas the remaining resource capacity can be represented by an upper and a lower service curve. As has been shown in [CKT03], the output arrival curves \( \alpha' \) can be computed by (4.9) and (4.10).

\[
\alpha''(\Delta) = \min \left\{ \sup_{\lambda > 0} \left\{ \inf_{0 \leq \mu < \lambda + \Delta} \{ \alpha''(\mu) + \beta''(\lambda + \Delta - \mu) - \beta''(\lambda) \} \right\}, \beta''(\Delta) \right\}
\]

(4.9)

\[
\alpha''(\Delta) = \min \left\{ \inf_{0 \leq \mu \leq \Delta} \left\{ \sup_{\lambda > 0} \{ \alpha''(\mu + \lambda) - \beta''(\lambda) \} + \beta''(\Delta - \mu) \right\}, \beta''(\Delta) \right\}
\]

(4.10)

Similarly, relations for the output service curves \( \beta' \) can be computed:

\[
\beta''(\Delta) = \inf_{\lambda \geq 0} \left\{ \beta''(\Delta + \lambda) - \bar{\alpha}'(\Delta + \lambda) \right\}
\]

(4.11)

\[
\beta''(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \left\{ \beta''(\Delta - \lambda) - \bar{\alpha}'(\Delta - \lambda) \right\}
\]

(4.12)

Local quantities describing a component’s performance can be derived analogously. Two parameters of interest are the backlog and delay experienced by events when being processed:

**Definition 4:** Let \( R(t) \) denote the number of events seen on the input stream of a GPC and \( R'(t) \) denote the number of events seen on the output stream of a GPC in the time interval \( 0 \leq \tau < t \). Then, the backlog \( b(t) \) and the delay \( d(t) \) are defined, as follows:

\[
b(t) = R(t) - R'(t)
\]

(4.13)

\[
d(t) = \inf \{ \tau \geq 0 : R(t) \leq R'(t + \tau) \}
\]

(4.14)

An upper bound of the maximum delay \( d_{\text{max}} \) experienced by an event and the maximum backlog (buffer fill level) \( b_{\text{max}} \) at a GPC are given by the following relations [LBT01]:

\[
b_{\text{max}} = \max_{t > 0} |b(t)| \leq B \left( \alpha'', \beta'' \right)
\]

(4.15)

\[
d_{\text{max}} = \max_{t > 0} |d(t)| \leq D \left( \alpha'', \beta'' \right)
\]

(4.16)

with

\[
B \left( \alpha'', \beta'' \right) = \sup_{\lambda \geq 0} \{ \alpha''(\lambda) - \beta''(\lambda) \}
\]

\[
D \left( \alpha'', \beta'' \right) = \sup_{\Delta \geq 0} \{ \inf_{\tau \geq 0} : \alpha''(\Delta) \leq \beta''(\Delta + \tau) \}
\]

(4.17)
4.3.5 Performance Analysis

So far, the modeling of event streams, computation and communication resources, as well as single software and hardware components has been described. In order to analyze the performance of an entire distributed system, one needs to build an abstract performance model of it. To obtain the analysis model of a system, all event streams are modeled by arrival curves, all computation and communication resources by service curves, as well as all actors in the system by GPCs. This way, one can analyze distributed systems with any number of shared computation and communication resources. Resource sharing policies currently supported by MPA include preemptive fixed priority [WTVL06], rate monotonic [WTVL06], time division multiple access [WT06b], earliest deadline first [WT06a], and first-come first-serve [PRT+10] scheduling. Fig. 18 illustrates different structures of the corresponding performance analysis models for several scheduling policies.

![Figure 18: Modeling of different scheduling policies in MPA. From left to right: preemptive fixed priority, earliest deadline first, and time division multiple access scheduling.](image)

By correctly interconnecting all arrival and service curves, one obtains the performance analysis model of a system, as shown in Fig. [16]. Global system properties, such as end-to-end delays, total buffer requirements, system throughput, and others, can be computed based on the local analysis of single components.

Tool support for MPA is available as a Matlab [Mat] toolbox [WT06c] that implements the basic operations from real-time calculus. Based on these operations, the Matlab toolbox provides methods for curve generation, implements the analysis for the scheduling policies mentioned above, and provides support for plotting arrival and service curves.
4.3.6 Min-Plus and Max-Plus Calculus

Network calculus and, consequently, also real-time calculus are based on min-plus calculus and its dual max-plus calculus \[BCOQ92, LBT01\]. In min-plus and max-plus calculus, the following basic operators are defined that will be used in the following sections.

Definition 5: (Min-Plus and Max-Plus Operators) For two wide-sense increasing functions \(f\) and \(g\), the min-plus convolution \((f \otimes g)\), the min-plus deconvolution \((f \oslash g)\), the max-plus convolution \((f \overset{\oslash}{\otimes} g)\), and the max-plus deconvolution \((f \overset{\otimes}{\oslash} g)\) are defined as follows:

\[
(f \otimes g)(\Delta) = \inf_{0 \leq \lambda \leq \Delta} \{ f(\Delta - \lambda) + g(\lambda) \}
\]

\[
(f \oslash g)(\Delta) = \sup_{\lambda \geq 0} \{ f(\Delta + \lambda) - g(\lambda) \}
\]

\[
(f \overset{\oslash}{\otimes} g)(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{ f(\Delta - \lambda) + g(\lambda) \}
\]

\[
(f \overset{\otimes}{\oslash} g)(\Delta) = \inf_{\lambda \geq 0} \{ f(\Delta + \lambda) - g(\lambda) \}
\]

Using these operators, the input–output relations (4.9)–(4.12) of a GPC can be compactly rewritten as follows:

\[
\alpha' = \min\{(\alpha^u \otimes \bar{\beta}^u) \otimes \bar{\beta}^l, \bar{\beta}^u\} \quad (4.18)
\]

\[
\alpha' = \min\{(\alpha^l \otimes \bar{\beta}^u) \otimes \bar{\beta}^l, \bar{\beta}^l\} \quad (4.19)
\]

\[
\beta'^u = (\beta^u - \bar{\alpha}^l) \overset{\oslash}{\otimes} 0 \quad (4.20)
\]

\[
\beta'^l = (\beta^l - \bar{\alpha}^u) \overset{\oslash}{\otimes} 0 \quad (4.21)
\]

Apart from these basic operators, the following lemma will be used later in this chapter:

Lemma 1: (Bounds on Differential Flow) Let \(R_1(t)\) and \(R_2(t)\) denote the number of events seen on the two input event streams in the time interval \(0 \leq \tau < t\) and let the arrival curves of these streams be denoted by \([\alpha^u_1, \alpha^l_1]\) and \([\alpha^u_2, \alpha^l_2]\), respectively. The differential flow, \(R_1(t) - R_2(s)\), \(\forall s \leq t\), can be bounded as follows:

\[
(\alpha^l_1 \overset{\oslash}{\otimes} \alpha^u_2)(t - s) \leq R_1(t) - R_2(s) \leq (\alpha^u_1 \otimes \alpha^l_2)(t - s) \quad (4.22)
\]

Proof. To prove (4.22), (4.1) is used to compute an upper and lower bound of \(R_1(t) - R_2(s)\).

\[
R_1(t) - R_2(s) = (R_1(t) - R_1(0)) - (R_2(s) - R_2(0)) \leq \alpha^u_1(t) - \alpha^l_2(s)
\]

\[
\leq \sup_{\lambda \geq 0} \{ \alpha^u_1(t + \lambda) - \alpha^l_2(s + \lambda) \} \quad (4.23)
\]

\[
\leq \sup_{\lambda \geq 0} \{ \alpha^u_1(t - s + \lambda) - \alpha^l_2(\lambda) \} = (\alpha^u_1 \otimes \alpha^l_2)(t - s)
\]
Chapter 4. Modeling Actors in Modular Performance Analysis

\[ R_1(t) - R_2(s) = (R_1(t) - R_1(0)) - (R_2(s) - R_2(0)) \]
\[ \geq a_1^l(t) - a_2^u(s) \]
\[ \geq \inf_{\lambda \geq 0} \{ a_1^l(t + \lambda) - a_2^u(s + \lambda) \} \]
\[ \geq \inf_{\lambda \geq 0} \{ a_1^l(t - s + \lambda) - a_2^u(\lambda) \} = (a_1^l \ominus a_2^u)(t - s) \]

4.4 Modeling Actors in MPA

This section presents a way in which actors with complex activation schemes can be modeled in the framework of MPA. Following the approach outlined in Section 4.2, this is done by using abstract OR and AND components as well as consumption and production curves to model single sub-actors. Afterwards, the analysis of non-preemptive fixed priority scheduling domains is presented.

4.4.1 Multiple Inputs and Outputs

In this section, the analysis of actors with multiple input and output streams is presented. Note that the basic approach as well as the abstract OR and AND components with two inputs have been introduced by Wandeler [Wan06, pp. 59–70]. Compared to [Wan06], however, the abstract OR and AND component are generalized to multiple inputs. Regarding the abstract AND component, a tighter lower output arrival curve is derived compared to [Wan06, (3.20) on p. 63].

4.4.1.1 Abstract OR Component

The abstract OR component produces an event at its output whenever an event is available on either of the input streams. For each output event, an event is removed from the corresponding input stream.

**Theorem 1:** (Abstract OR Component) Assume an abstract OR component with \( N \) input event streams that are modeled as abstract event streams with...
arrival curves $\alpha_i = [\alpha_i^u, \alpha_i^l]$, $1 \leq i \leq N$. Then, the output arrival curve $\alpha_{OR} = [\alpha_{OR}^u, \alpha_{OR}^l]$ is given by:

$$\alpha_{OR}^u = \sum_{i=1}^{N} \alpha_i^u$$  \hspace{1cm} (4.25) $$\alpha_{OR}^l = \sum_{i=1}^{N} \alpha_i^l$$  \hspace{1cm} (4.26)

Moreover, no delays or backlogs occur at an abstract OR component:

$$d_{i_{\text{max}}} = 0$$  \hspace{1cm} (4.27) $$b_{i_{\text{max}}} = 0$$  \hspace{1cm} (4.28)

**Proof.** (4.25) and (4.26) are proved by computing the difference between the number of events at the inputs and the output of the OR component and bounding it using (4.1). Let $R_i(t)$, $1 \leq i \leq N$, denote the number of events seen on the input event streams in the time interval $0 \leq \tau < t$. The number of events seen on the output stream $R_{OR}(t)$ of the OR component in the time interval $0 \leq \tau < t$ is given by:

$$R_{OR}(t) = \sum_{i=1}^{N} R_i(t)$$  \hspace{1cm} (4.29)

Thus, for all $s \leq t$,

$$R_{OR}(t) - R_{OR}(s) = \sum_{i=1}^{N} R_i(t) - \sum_{i=1}^{N} R_i(s) = \sum_{i=1}^{N} R_i(t) - R_i(s)$$  \hspace{1cm} (4.30)

By replacing $R_i(t) - R_i(s)$ with their upper and lower bounds, the following inequality is obtained that proves (4.25) and (4.26) by the definition of the arrival curve (4.1).

$$\sum_{i=1}^{N} \alpha_i^l(t-s) \leq R_{OR}(t) - R_{OR}(s) \leq \sum_{i=1}^{N} \alpha_i^u(t-s)$$  \hspace{1cm} (4.31)

(4.27) and (4.28) directly follow from the definition of the abstract OR component and that $R(t)$ is wide-sense increasing:

$$b(t) = R(t) - R'(t) = \sum_{i=1}^{N} R_i(t) - R_{OR}(t) = \sum_{i=1}^{N} R_i(t) - \sum_{i=1}^{N} R_i(t) = 0$$  \hspace{1cm} (4.32)
\[ d(t) = \inf \{ \tau \geq 0 : R(t) \leq R'(t + \tau) \} \]
\[ = \inf \left\{ \tau \geq 0 : \sum_{i=1}^{N} R_i(t) \leq R_{\text{OR}}(t + \tau) \right\} \]
\[ = \inf \left\{ \tau \geq 0 : \sum_{i=1}^{N} R_i(t) \leq \sum_{i=1}^{N} R_i(t + \tau) \right\} = 0 \]
(4.33)

**4.4.1.2 Abstract AND Component**

![Figure 20: Abstract AND component.](image)

The abstract AND component produces an event at its output whenever at least one event is available on all input event streams. For each output event, one event is removed from all input streams.

**Theorem 2: (Abstract AND Component)** Assume an abstract AND component with \( N \) input event streams that are modeled as abstract event streams with arrival curves \( \alpha_i = [\alpha_u^i, \alpha_l^i], 1 \leq i \leq N \), and let \( B_0^i \) denote the initial buffer fill level at input \( i \). Then, the output arrival curve \( \alpha_{\text{AND}} = [\alpha_u^{\text{AND}}, \alpha_l^{\text{AND}}] \) is given by:

\[
\alpha_u^{\text{AND}} = \max_{1 \leq i \leq N} \left\{ \min \left\{ \min_{1 \leq j \leq N} \left\{ \alpha_u^i \ominus \alpha_l^j + B_0^j - B_0^i \right\}, \alpha_u^i \right\} \right\}
\]  
(4.34)

\[
\alpha_l^{\text{AND}} = \min_{1 \leq i \leq N} \left\{ \max \left\{ \max_{1 \leq j \leq N} \left\{ \alpha_l^i \ominus \alpha_u^j + B_0^j - B_0^i \right\}, \alpha_l^i \right\} \right\}
\]  
(4.35)

Moreover, let \( d_{\text{max}}^i \) and \( b_{\text{max}}^i \), \( 1 \leq i \leq N \) denote the delay and backlog bounds for events on the input streams. Then,

\[
d_{\text{max}}^i = \max_{1 \leq j \leq N} \left\{ D \left( \alpha_u^i + B_0^i, \alpha_l^j + B_0^j \right) \right\}
\]  
(4.36)

\[
b_{\text{max}}^i = \max_{1 \leq j \leq N} \left\{ B \left( \alpha_u^i + B_0^i, \alpha_l^j + B_0^j \right), 0 \right\}
\]  
(4.37)
Thus, for all $s \leq t$,
\[
R_{\text{AND}}(t) = \min_{1 \leq i \leq N} \left\{ R_i(t) + B_i^0 \right\}
\] (4.38)

One can now compute an upper and a lower bound for (4.39) by using (4.22) and (4.40).
\[
R_{\text{AND}}(t) - R_{\text{AND}}(s) = \min_{1 \leq i \leq N} \left\{ R_i(t) + B_i^0 \right\} - \min_{1 \leq i \leq N} \left\{ R_i(s) + B_i^0 \right\}
\] (4.39)

Observe that for two sets $\mathcal{A} = a_1, \ldots, a_N \subset \mathbb{R}$, $|\mathcal{A}| = N$, and $\mathcal{B} = b_1, \ldots, b_M \subset \mathbb{R}$, $|\mathcal{B}| = M$, the following equations apply:
\[
\min_{1 \leq i \leq N} \left\{ a_i \right\} - \min_{1 \leq j \leq M} \left\{ b_j \right\} = \max_{1 \leq i \leq N} \left\{ \min_{1 \leq j \leq M} \left\{ a_i - b_j \right\} \right\} = \min_{1 \leq i \leq N} \left\{ \max_{1 \leq j \leq M} \left\{ a_i - b_j \right\} \right\}
\] (4.40)

Proof. To prove (4.34) and (4.35), the differential flow after the AND component is computed and bounded using (4.22). Let $R_1(t)$ and $R_2(t)$ denote the number of events seen on the two input event streams in the time interval $0 \leq \tau < t$. The number of events seen on the output stream $R_{\text{AND}}(t)$ of the AND component in the time interval $0 \leq \tau < t$ is given by
\[
R_{\text{AND}}(t) = \min_{1 \leq i \leq N} \left\{ R_i(t) + B_i^0 \right\}
\] (4.38)

Thus, for all $s \leq t$,
\[
R_{\text{AND}}(t) - R_{\text{AND}}(s) = \min_{1 \leq i \leq N} \left\{ R_i(t) + B_i^0 \right\} - \min_{1 \leq i \leq N} \left\{ R_i(s) + B_i^0 \right\} \] (4.39)
Next, the proof of the delay bound (4.36) follows. Using the definition of the delay bound (4.14) and a lower bound on the differential flow (4.22), (4.36) can be derived as follows:

\[
d_i(t) = \inf \left\{ \tau \geq 0 : R_i(t) + B_i^0 \leq \min_{1 \leq j \leq N} \{ R_j(t + \tau) + B_j^0 \} \right\} \\
= \max_{1 \leq j \leq N, j \neq i} \left\{ \inf \left\{ \tau \geq 0 : R_i(t) + B_i^0 \leq R_j(t + \tau) + B_j^0 \right\} \right\} \\
= \max_{1 \leq j \leq N, j \neq i} \left\{ \inf \left\{ \tau \geq 0 : 0 \leq R_j(t + \tau) - R_i(t) + B_j^0 - B_i^0 \right\} \right\} \\
\leq \max_{1 \leq j \leq N, j \neq i} \left\{ \inf \left\{ \tau \geq 0 : 0 \leq \left( \alpha_j^l \odot \alpha_i^u \right)(\tau) + B_j^0 - B_i^0 \right\} \right\} \\
= \max_{1 \leq j \leq N, j \neq i} \left\{ \inf \left\{ \tau \geq 0 : 0 \leq \alpha_j^u(\tau + \lambda) - \alpha_i^u(\lambda) + B_j^0 - B_i^0 \right\} \right\} \\
= \max_{1 \leq j \leq N, j \neq i} \left\{ \sup_{\Delta \geq 0} \left\{ \inf \left\{ \tau \geq 0 : \alpha_i^u(\Delta) + B_i^0 \leq \alpha_j^u(\tau + \Delta) + B_j^0 \right\} \right\} \right\} \\
\tag{4.43}
\]

Using the definition of the backlog bound (4.13) and an upper bound on the differential flow (4.22), (4.37) can be derived as follows:

\[
b_i(t) = R_i(t) + B_i^0 - \min_{1 \leq j \leq N} \{ R_j(t) + B_j^0 \} \\
= \max_{1 \leq j \leq N, j \neq i} \left\{ R_i(t) - R_j(t) + B_i^0 - B_j^0, 0 \right\} \\
\leq \max_{1 \leq j \leq N, j \neq i} \left\{ \left( \alpha_i^u \odot \alpha_j^l \right)(0) + B_i^0 - B_j^0, 0 \right\} \tag{4.44}
\]

\[
= \max_{1 \leq j \leq N, j \neq i} \left\{ \sup_{\lambda \geq 0} \left\{ \alpha_j^u(\lambda) - \alpha_i^u(\lambda) + B_i^0 - B_j^0 \right\}, 0 \right\} \\
= \max_{1 \leq j \leq N, j \neq i} \left\{ \sup_{\lambda \geq 0} \left\{ \alpha_j^u(\lambda) + B_i^0 - \left( \alpha_i^u(\lambda) + B_i^0 \right) \right\}, 0 \right\}
\]
One can observe that the lower arrival curve \(4.19\) is a larger and, therefore, a tighter bound compared to the bound obtained by Wandeler \([\text{Wan06}, (3.20)\] on p. 63), that is:

\[
\tilde{\alpha}^l_{\text{AND}} = \max \left\{ \min \left\{ \alpha^l_1 \ominus \alpha^u_2 + B^0_1 - B^0_2, \alpha^l_2 \right\}, \min \left\{ \alpha^l_2 \ominus \alpha^u_1 + B^0_2 - B^0_1, \alpha^l_1 \right\} \right\} \tag{4.45}
\]

**Proof.** For the special case of two inputs, the bound derived in this thesis is given by:

\[
\alpha^l_{\text{AND}} = \min \left\{ \max \left\{ \alpha^l_1 \ominus \alpha^u_2 + B^0_1 - B^0_2, \alpha^l_1 \right\}, \max \left\{ \alpha^l_2 \ominus \alpha^u_1 + B^0_2 - B^0_1, \alpha^l_2 \right\} \right\} \tag{4.46}
\]

<table>
<thead>
<tr>
<th>Table 4:</th>
<th>Point-wise comparison of (\tilde{\alpha}^l_{\text{AND}}) and (\alpha^l_{\text{AND}}), showing that (\alpha^l_{\text{AND}} \geq \tilde{\alpha}^l_{\text{AND}}).</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &lt; B</td>
<td>C &lt; D</td>
</tr>
<tr>
<td>A &lt; D</td>
<td>(\tilde{\alpha}^l_{\text{AND}}) max ([A, C]) C</td>
</tr>
<tr>
<td>A ≥ D</td>
<td>(\alpha^l_{\text{AND}}) min ([B, D]) A</td>
</tr>
<tr>
<td>C &lt; B</td>
<td>D</td>
</tr>
<tr>
<td>C ≥ B</td>
<td>D</td>
</tr>
<tr>
<td>A &lt; D</td>
<td>A</td>
</tr>
<tr>
<td>A ≥ D</td>
<td>(\tilde{\alpha}^l_{\text{AND}}) max ([A, B]) D</td>
</tr>
<tr>
<td>C &lt; B</td>
<td>(\alpha^l_{\text{AND}}) min ([C, D]) (\times)</td>
</tr>
<tr>
<td>C ≥ B</td>
<td>A</td>
</tr>
<tr>
<td>A &lt; D</td>
<td>(\tilde{\alpha}^l_{\text{AND}}) max ([A, C]) C</td>
</tr>
<tr>
<td>A ≥ D</td>
<td>(\alpha^l_{\text{AND}}) min ([B, D]) A</td>
</tr>
<tr>
<td>C &lt; B</td>
<td>D</td>
</tr>
<tr>
<td>C ≥ B</td>
<td>D</td>
</tr>
<tr>
<td>A &lt; D</td>
<td>A</td>
</tr>
<tr>
<td>A ≥ D</td>
<td>(\tilde{\alpha}^l_{\text{AND}}) max ([A, B]) D</td>
</tr>
<tr>
<td>C &lt; B</td>
<td>(\alpha^l_{\text{AND}}) min ([C, D]) (\times)</td>
</tr>
<tr>
<td>C ≥ B</td>
<td>A</td>
</tr>
<tr>
<td>A ≥ D</td>
<td>(\tilde{\alpha}^l_{\text{AND}}) max ([A, B]) D</td>
</tr>
<tr>
<td>C &lt; B</td>
<td>(\alpha^l_{\text{AND}}) min ([C, D]) (\times)</td>
</tr>
<tr>
<td>C ≥ B</td>
<td>A</td>
</tr>
</tbody>
</table>

By point-wise comparison of \(4.45\) and \(4.46\), it can be shown that \(\alpha^l_{\text{AND}} \geq \tilde{\alpha}^l_{\text{AND}}\). Let \(A = \alpha^l_1 \ominus \alpha^u_2 + B^0_1 - B^0_2\), \(B = \alpha^l_2\), \(C = \alpha^l_2 \ominus \alpha^u_1 + B^0_2 - B^0_1\), and \(D = \alpha^l_1\), such that \(\tilde{\alpha}^l_{\text{AND}} = \max\{\min\{A, B\}, \min\{C, D\}\}\) and \(\alpha^l_{\text{AND}} = \min\{\max\{A, D\}, \max\{C, B\}\}\). There are 24 different possibilities for numerically ordering \(A, B, C, D\). These possibilities are listed in Table 4 which shows that \(4.46\) is greater than (or equal to) \(4.45\), indeed. Note
that due to the definition of max-plus deconvolution, \( A \geq D \) and \( C \geq B \) cannot both be fulfilled at the same time which voids the cases indicated by \( \times \) in the table.

\[ \square \]

### 4.4.1.3 GPC with Boolean Activation Condition

A GPC with a Boolean activation condition is analyzed by modeling the activation condition as a network of abstract OR and AND components and using the sole output of this network as the input for the GPC. Specifically, every Boolean combination of OR and AND (excluding negations) can be rewritten as an expression in disjunctive normal form. This corresponds to a 2-layer network of abstract OR and AND components where the first layer consists only of AND components and the second layer is a single OR component, as sketched in Fig. 21.

![Figure 21: Principle of analyzing actors processing multiple input streams.](image)

To analyze the entire network, first the input arrival curves are propagated to the single input of the GPC using the expressions derived in this chapter. Then, the output arrival and service curves are computed using the standard input–output relations of the GPC (4.18)–(4.21). Similarly, the delay and backlog bounds for the individual input streams are computed by summing up the backlogs and delays experienced across the different components. The total buffer requirement for a component is then computed by summing up the individual backlogs. Note that, in general, these bounds are not tight because the event trace leading to the worst-case delay and backlog at one point in the analysis
model does not coincide with the trace leading to the worst-case delay and backlog at another point in the analysis model, and hence adding up these worst-case quantities is too pessimistic.

So far, it has been assumed that every event leads to one activation of an actor. This is not necessarily the case in a real system where an actor might read multiple tokens per firing. The actual number of tokens might be fixed, as in synchronous dataflow, might vary in a cyclic pattern, as in cyclo-static dataflow, or might vary in a more general way. As long as this variation is bounded, it can be modeled using the so-called consumption curves proposed in [MZCW04], see Fig. 22. Specifically, the concatenation $\kappa(\alpha)$ "normalizes" an arrival curve in the sense that in the resulting arrival curve one event corresponds to one activation and the variation of consumed tokens is factored in.

![Figure 22: Consumption curve. $\kappa = [\kappa^u, \kappa^l]$ gives upper and lower bounds on the number of activations caused by a number of consecutive events. The concatenation $\kappa(\alpha)$ gives the number of activations per time interval.](image)

**Definition 6: (Consumption Curve)** Upper and lower consumption curves $\kappa = [\kappa^u, \kappa^l]$ describe how many activations of a component are caused by $e$ consecutive events.

Consequently, each input arrival curve is concatenated with the corresponding consumption curve to model the variation of the consumed tokens, as shown in Fig. 21. Note that as a consequence of applying $\kappa$, any $e$ consecutive events at the input of subsequent components correspond to up to $(\kappa^u)^{-1}(e)$ events of the original arrival curve. In particular, this means that the backlog bound needs to be adjusted accordingly, as follows. Let $\bar{B}$ denote the backlog bound computed for an input stream $\alpha$ after concatenation with the consumption curve $\kappa$. Then, the backlog bound for that input is given by $B = (\kappa^u)^{-1}(\bar{B})$. 
4.4.1.4 Multiple Outputs

The analysis of actors that send data to multiple output streams is straightforward. Basically, multiple outputs can be modeled simply by duplicating the output arrival curve of the GPC. Only when multiple sub-actors send data to the same output stream or when the number of tokens produced in one activation does not equal one, additional modeling effort is necessary.

To model output streams to which events are contributed by multiple sub-actors, the abstract OR component introduced earlier is used because its semantics also match what happens when multiple actors share a single output stream: whenever an actor writes a token to an output stream, the token is immediately committed to the corresponding FIFO channel. Likewise, an abstract OR component immediately forwards any event occurring on any input stream to the joint output stream. Therefore, the upper and lower arrival curve for an output event stream that is accessed by multiple actors is given by the sum of the upper and lower output arrival curves of these actors, see (4.25)–(4.26).

For modeling a possible variation in the number of produced tokens, one can define so-called production curves [MZCW04] analogously to consumption curves.

Definition 7: (Production Curve) An upper and lower production curve \( \pi = [\pi^u, \pi^l] \) describe how many events are produced at most and at least in \( e \) consecutive activations of a component.

By concatenating the single output arrival curve of a GPC with a production curve, one can factor in the variation of produced tokens for each output stream, as shown in Fig. 15.

4.4.2 Non-Preemptive Fixed Priority Scheduling

In Section 4.2, it has been pointed out that an actor structured like the one in Listing 5 can be viewed as a non-preemptive fixed priority scheduling domain because the order in which the activation conditions of the individual sub-actors are checked imposes priorities on the sub-actors: any sub-actor whose activation condition is checked earlier than the activation condition of others will execute as long as its activation condition is fulfilled and, consequently, prevent other sub-actors from executing. The following theorem describes the transfer functions of the component shown in Fig. 23 which models a non-preemptive fixed priority scheduling domain.
Figure 23: MPA model of a non-preemptive fixed priority scheduling domain.

Theorem 3: (Non-Preemptive Fixed Priority Scheduling) Assume given a non-preemptive fixed priority scheduling domain comprising $N$ GPCs that are ordered according to their priorities ($1$ is highest and $N$ is lowest). Let $\alpha_i$ and $\alpha'_i$, $1 \leq i \leq N$, denote the input and output arrival curves of these GPCs and let WCED$_i$ denote the worst-case execution demand of the $i$-th GPC (in clock cycles). Furthermore, let $\beta$ and $\beta'$ denote the input and output service curves of the scheduling domain. Then, the input–output relations for a non-preemptive fixed priority scheduling domain are given by (4.47)–(4.50).

$$\alpha'^u_i = \min \left\{ (\alpha^u_i \otimes \bar{\beta}^u) \otimes \bar{\beta}'_1, \bar{\beta}^u \right\}$$  (4.47)

$$\alpha'^l_i = \min \left\{ (\alpha'_i \otimes \bar{\beta}^u) \otimes \bar{\beta}'_i, \bar{\beta}'_i \right\}$$  (4.48)

$$\beta'^u = \left( \beta'^u - \sum_{i=1}^{N} \bar{\alpha}'_i \right) \overline{\otimes} 0$$  (4.49)

$$\beta'^l = \left( \beta'^l - \sum_{i=1}^{N} \bar{\alpha}^u_i \right) \overline{\otimes} 0$$  (4.50)

with

$$\beta'^u_i = \min \left\{ \beta'^u_i, \left( \beta'^u - \sum_{j=1}^{i-1} \bar{\alpha}'_j \right) \overline{\otimes} 0 \right\} + \text{WCED}_i \right\}$$,  \hspace{1cm} \beta'^u_i = (\gamma^u)^{-1}(\beta'^u_i) \hspace{1cm} (4.51)

$$\beta'^l_i = \max \left\{ 0, \left( \beta'^l - \max_{1 \leq j \leq N} \{ \text{WCED}_j \} - \sum_{j=1}^{i-1} \bar{\alpha}^u_j \right) \overline{\otimes} 0 \right\}$$,  \hspace{1cm} \beta'^l_i = (\gamma^u)^{-1}(\beta'^l_i) \hspace{1cm} (4.52)

Proof. First, conventional preemptive fixed priority scheduling is considered. In this case, the service available to a GPC with priority
$i$ is bounded by the following service curves which can be derived by iteratively applying (4.20) and (4.21).

\[
\beta_{i}^{u, \text{preemptive}} = \left(\beta^{u} - \sum_{j=1}^{i-1} \bar{\alpha}_{j}^{l}\right) \odot 0
\]

\[
\beta_{i}^{l, \text{preemptive}} = \left(\beta^{l} - \sum_{j=1}^{i-1} \bar{\alpha}_{j}^{u}\right) \odot 0
\]  

(4.53)

Now (4.51) and (4.52) follow by considering the difference between non-preemptive and preemptive fixed priority scheduling: concerning the best case, it can be observed that once a GPC is activated, it is permitted to complete one firing which will occupy the resource at most by its worst-case execution demand. This additional resource availability is considered by adding the corresponding term in (4.51). Concerning the worst case, a GPC can be prevented from accessing the resource by at most the maximum worst-case execution demand of a GPC. This reduction of the resource availability in a non-preemptive compared to a preemptive fixed priority scheduling domain is considered by subtracting the corresponding term in (4.52). To exclude the cases which, due to this shifting, have a resulting upper service curve that exceeds the original upper service curve or have a lower service curve that drops below 0, the according minimum and maximum are taken, respectively.

\[\square\]

Note that the upper and lower service curves (4.51)–(4.52) are, in general, not tight. Considering a sufficiently long interval (the hyper-period of a periodic set of actors, for instance), the service available to an actor of a certain priority is the same in preemptive and non-preemptive fixed priority scheduling. This is not the case for (4.51)–(4.52) when compared to (4.53), however.

### 4.5 Experimental Results

In this chapter, a method for the performance analysis of actors with complex activation schemes has been proposed in the framework of Mpa. In this section, this method will be applied to analyze several systems. The aim is to show that the accuracy of the obtained results is similar to alternative analysis methods without sacrificing the advantages of Mpa, namely compositionality and a powerful event stream and resource model. Note that it is difficult to give general statements, however,
because a fair evaluation or comparison of best-case/worst-case analysis methods is a delicate issue. Perathoner et al. [PWT+09], for instance, conclude that already for a simple system consisting of three actors on two processors, no general statement about the accuracy (tightness) of different methods can be given.

In the following, OR- and AND-activation are considered separately from non-preemptive fixed priority scheduling. For both aspects, multiprocessor (sub-)systems (rather than single-chip multiprocessors) reported in the related literature are analyzed.

### 4.5.1 OR- and AND-Activation

The considered subsystems are the OR- and AND-activated actors shown in Fig. 24. These subsystems were analyzed in [JRE05] using SymTA/S where activation schemes are also modeled by dedicated OR and AND components.

![Figure 24: OR- and AND-activated actors.](image)

#### 4.5.1.1 OR-Activation

The input streams S1, S2, and S3 comply to the sporadic event model with periods of 1000, 750, and 600 time units, respectively. In Fig. 25, the corresponding input arrival curves are shown.

![Figure 25: Input arrival curves of the OR-activated actor in Fig. 24.](image)

In Fig. 26, the output arrival curve obtained using (4.25)–(4.26) is plotted and compared to the curve obtained by SymTA/S [JRE05]. One can see that the arrival curve obtained by MPA is tighter than the curve obtained by SymTA/S. Unlike MPA, the arrival curve of the OR component
needs to be approximated by a standard event model in SymTA/S. Since this approximation needs to be conservative, a gap between the curves obtained by MPA and SymTA/S emerges, as shown in Fig. 26.

Figure 26: Output arrival curves of abstract OR component (left) and abstract AND component (right) in Fig. 24.

4.5.1.2 AND-Activation

The input streams $S_4$, $S_5$, and $S_6$ in Fig. 24 correspond to the periodic-with-jitter event model with a period of 4 and a jitter of 0, 2, and 3 time units. The number of events per period in $S_4$, $S_5$, and $S_6$ is 2, 2, and 3 which is also the number of events required on those streams to activate the actor. In Fig. 27, the corresponding input arrival curves and consumption curves are shown.

Figure 27: Input arrival and consumption curves of the AND-activated actor in Fig. 24.
The resulting output arrival curves obtained by applying (4.34)–(4.35) are plotted in Fig. 26. In this example, the output arrival curve corresponds to the periodic-with-jitter event model with a period of 4 and a jitter of 3 time units which can be exactly represented by both methods, \( \text{MPa} \) and SymTA/S. Also, the delays and backlogs computed by the two methods coincide: the delays for the streams \( S_4, S_5, \) and \( S_6 \) are 7, 9, and 9 time units and the backlogs 4, 6, and 9 tokens, respectively.

### 4.5.2 Non-Preemptive Fixed Priority Scheduling

Due to its simplicity, non-preemptive fixed priority scheduling is widely adopted in embedded systems. One example is the controller area network (CAN) which is a serial bus system for embedded control using a non-preemptive fixed priority arbitration. Standardized as ISO 11898-1 in 1993, more than two billion CAN enabled nodes have been sold [CAN].

CAN is basically a broadcast serial bus where each node connected to the bus can both send and receive messages (but not simultaneously). CAN is non-preemptive because nodes can only start transmitting messages when the bus is free whereby a message consists of an ID and the payload data. When multiple nodes start transmitting at the same time, the message ID is used to resolve collisions: the message with lowest ID has highest priority, that is, any node which detects that a message with a lower ID is being sent will cease transmitting its message.

#### 4.5.2.1 Network of Electronic Control Units

The considered system is a network of electronic control units (ECU) communicating via a CAN bus, see Fig. 28 and Table 5. It consists of three ECUs that are connected by a shared CAN bus and are executing a distributed application. To schedule the actors on the ECUs, preemptive fixed priority scheduling is used. Any scheduling overhead on both the ECUs and the CAN bus is neglected.

Originally considered in [NPGSV07], Chokshi et al. [CB08] present results derived by their own method based on real-time calculus and results derived by applying holistic analysis [GH98, BLV07]. These results are reproduced in Table 6 along with the results derived by applying the analysis presented in this chapter. Specifically, (4.53) is used to analyze the ECUs, and (4.51)–(4.52) are used to analyze the CAN bus. One can see that the obtained results are equal except for two cases, indicating that the method proposed in this chapter has a similar accuracy compared to the two other methods.
Chapter 4. Modeling Actors in Modular Performance Analysis

Figure 28: *M*PA model of a network of ECUs.

Table 5: System parameters of the network of ECUs depicted in Fig. 28

<table>
<thead>
<tr>
<th>ACTOR</th>
<th>EX. TIME</th>
<th>PRIORITY¹</th>
<th>CHANNEL</th>
<th>EX. TIME</th>
<th>PRIORITY¹</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ECU 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>8</td>
<td>1</td>
<td>T1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>T8</td>
<td>14</td>
<td>2</td>
<td>M2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>T9</td>
<td>2</td>
<td>3</td>
<td>M4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>ECU 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>8</td>
<td>1</td>
<td>M7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>T6</td>
<td>6</td>
<td>2</td>
<td>M10</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>T11</td>
<td>6</td>
<td>3</td>
<td>M12</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td><strong>ECU 3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>6</td>
<td>1</td>
<td>S1</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>T13</td>
<td>8</td>
<td>1</td>
<td>S2</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td><strong>STREAM</strong></td>
<td><strong>PERIOD</strong></td>
<td></td>
<td><strong>S1</strong></td>
<td><strong>S2</strong></td>
<td><strong>S3</strong></td>
</tr>
<tr>
<td><strong>S1</strong></td>
<td>15</td>
<td>1</td>
<td>S1</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td><strong>S2</strong></td>
<td>40</td>
<td>2</td>
<td>S2</td>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td><strong>S3</strong></td>
<td>30</td>
<td>3</td>
<td>S3</td>
<td>30</td>
<td>3</td>
</tr>
</tbody>
</table>

¹ A lower numerical value corresponds to a higher priority.
Table 6: Comparison of results derived by holistic analysis and a method based on real-time calculus with results derived by the method introduced in this chapter.

<table>
<thead>
<tr>
<th>GPC</th>
<th>HOLISTIC, REAL-TIME CALCULUS</th>
<th>MODULAR PERFORMANCE ANALYSIS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DELAY</td>
<td>BACKLOG</td>
</tr>
<tr>
<td>T1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>M2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>T3</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>M4</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>T5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>T6</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>M7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>T8</td>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td>T9</td>
<td>130</td>
<td>5</td>
</tr>
<tr>
<td>M10</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>T11</td>
<td>104</td>
<td>6</td>
</tr>
<tr>
<td>M12</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>T13</td>
<td>44</td>
<td>3</td>
</tr>
</tbody>
</table>

<sup>1</sup> Jitter of the input stream.

<sup>2</sup> Jitter of the input stream. Given an originally periodic event stream with period \( P \) and an arrival curve \( \alpha \) describing the timing of that event stream after being processed, the jitter is computed by \((D(\alpha^u, \alpha^l) - P)/2\).

4.5.2.2 Automotive System

The considered system is an automotive system adopted from [RHE07], see Fig. 29 and Table 7. The system consists of two independently developed subsystems, each of which consists of two ECUs and a CAN bus. These subsystems need to be integrated into one system by merging the two CAN buses. Except for ECU 1, all ECUs are scheduled according to a preemptive fixed priority scheduling policy. On ECU 1, the ERCOSEK real-time operating system is executed, which allows for preemptive and priority-based cooperative scheduling. (Priority-based cooperative scheduling corresponds to non-preemptive fixed priority scheduling with the difference that an actor might voluntarily suspend execution to enable the execution of actors of higher priority.) In the given system, two high-priority actors are scheduled in a preemptive manner and two actors of lower priority are cooperatively scheduled on ECU 1. Again, any scheduling overhead is neglected.

To analyze the system using \( MPA \), one can proceed as follows: the CAN bus is considered as a non-preemptive fixed priority scheduling domain.
like in the previous system. ECU 2 to 4 are modeled as preemptive fixed priority scheduling domains \((4.53)\). ECU 1 is modeled as a hierarchical scheduling domain where preemptive fixed priority scheduling is applied for the two high-priority actors \((4.53)\) and the relations for non-preemptive fixed priority scheduling \((4.51)–(4.52)\) are used to derive the service curves for the other two actors. This is possible because the arguments used to derive \((4.51)–(4.52)\) equally apply to priority-based cooperative scheduling. The results will be pessimistic concerning the worst-case resource availability, however, because cooperative actors possess
### Table 7: System parameters of the automotive system depicted in Fig. 29

<table>
<thead>
<tr>
<th>ACTOR</th>
<th>EX. TIME</th>
<th>PRIORITY¹</th>
<th>CHANNEL</th>
<th>EX. TIME</th>
<th>PRIORITY²</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU 1</td>
<td></td>
<td></td>
<td>CAN 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₀</td>
<td>[0.2, 0.3]</td>
<td>10</td>
<td>C₀</td>
<td>[1.08, 1.32]</td>
<td>1</td>
</tr>
<tr>
<td>T₁</td>
<td>[0.2, 0.3]</td>
<td>5</td>
<td>C₁</td>
<td>[0.76, 0.92]</td>
<td>3</td>
</tr>
<tr>
<td>T₂</td>
<td>[0.1, 0.2]</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₃</td>
<td>[1.2, 2.3]</td>
<td>1</td>
<td>C₂</td>
<td>[0.60, 0.72]</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C₃</td>
<td>[0.68, 0.82]</td>
<td>2</td>
</tr>
<tr>
<td>ECU 2</td>
<td></td>
<td></td>
<td>CAN 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₄</td>
<td>[0.7, 0.8]</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₅</td>
<td>[0.1, 0.2]</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₇</td>
<td>[0.1, 0.6]</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECU 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₈</td>
<td>[0.3, 0.4]</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₉</td>
<td>[1.0, 2.0]</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁₁</td>
<td>[3.0, 5.0]</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECU 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₆</td>
<td>[1.1, 1.5]</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁₀</td>
<td>[0.3, 0.6]</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ For ECU 1, a higher numerical value corresponds to a higher priority. For all other resources, a lower numerical value corresponds to a higher priority.

---

four preemption points which shortens the maximum time a high-priority actor needs to wait. At the same time, the results will be optimistic concerning the best-case resource availability because the maximum time a low-priority actor can continue running is also shortened.

The quantities of interest in this system are the end-to-end delays of events on the different streams. In [RHE07], the original system with two separate CAN busses as well as the integrated system with a single CAN bus were analyzed using SymTA/S. These results are reproduced in Table 8 along with the results derived by applying the analysis presented in this chapter. Note that in order to obtain a bound on the end-to-end delays, one could simply add up the delays of events at the single GPCs. In Mpa, however, one can use a principle from network calculus known as “pay bursts only once” [LBT01] which states that a single event cannot experience the worst-case at all GPCs when propagating through a chain of GPCs. This allows obtaining tighter bounds on delays and backlogs of event streams propagating through several GPCs. Therefore,
this principle has been applied to obtain the bounds listed in Table 8.

Table 8 shows that the results of both analysis methods are similar. In some cases, SymTA/S yields lower bounds whereas in other cases MPA yields lower bounds. The cases where SymTA/S yields lower bounds can be attributed to the analysis of ECU 1 because SymTA/S integrates special analysis methods for analyzing ERCOSEK systems. In addition, the service curves for modeling non-preemptive fixed priority scheduling in MPA are not tight, hence these bounds are overestimated. On the other hand, the cases where MPA yields lower bounds can be attributed to the greater generality of the event model of MPA compared to SymTA/S.

Table 8: Comparison of analysis results of SymTA/S and MPA for the automotive system in Fig. 29.

<table>
<thead>
<tr>
<th>PATH</th>
<th>SYM</th>
<th>MPA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPLIT CANs</td>
<td>SINGLE CAN</td>
</tr>
<tr>
<td>$S_2 \rightarrow S_5$</td>
<td>12.01</td>
<td>20.18</td>
</tr>
<tr>
<td>$S_9 \rightarrow S_{11}$</td>
<td>5.12</td>
<td>7.92</td>
</tr>
<tr>
<td>$S_3 \rightarrow S_8$</td>
<td>9.92</td>
<td>33.09</td>
</tr>
</tbody>
</table>

4.6 Related Work

Methods to analyze actors with OR- and AND-activation as well as non-preemptive fixed priority scheduling domains have not only been considered in MPA but also in other frameworks for best-case/worst-case performance analysis.

OR- and AND-activation have been considered in [JRE05] in the context of SymTA/S. In an open-source framework for holistic scheduling analysis, namely MAST (modeling and analysis suite for real-time applications), OR- and AND-activation can be modeled using so-called concentrators and barriers [HGGM01]. These methods are based on standard event models which, in general, cannot exactly represent the timing behavior resulting from Boolean activation conditions. As shown in the previous section, this requires making approximations, which is not the case in MPA because arrival curves can tightly bound any event stream with non-deterministic timing behavior.

The analysis of non-preemptive fixed priority scheduling domains is related to the analysis of non-preemptive fixed priority multiplexers in network queuing theory. In this context, delay and backlog bounds for a non-preemptive fixed priority multiplexer with constant output
rate are derived by Cruz in [Cru91a]. Furthermore, non-preemptive fixed priority queuing of two streams has been considered by Le Boudec and Thiran in [LBT01, pp. 20–21] and of multiple streams in [Sch03] by Schmitt. In [BLV07], non-preemptive fixed priority scheduling domains are considered in the context of holistic analysis. In contrast to Mpa, these methods neither provide lower bounds on the number of events at the output of an actor nor upper bounds on the amount of the remaining service. In [CD08], a method based on real-time calculus is presented in which upper and lower service curves are computed, as well. Compared to the closed-form solution presented in this chapter, the service curves are computed following an iterative procedure. Therefore, that method likely is more computation-intensive than the approach presented in this chapter, in exchange for yielding tighter bounds.

Apart from this directly related work, complementary methods have been proposed in the framework of Mpa that are useful for analyzing actors with multiple inputs and outputs. These methods consider additional information about event streams to obtain tighter bounds compared to the analysis presented in this chapter. In [HTS07], for instance, split-and-merge structures are analyzed where multiple streams originating from the same actor are later merged at an actor. This leads to timing correlations between the input event streams of the latter which can be utilized to obtain tighter bounds. Join-and-fork structures where multiple event streams are joined into a single event stream and forked into the original streams at later points have been considered in [PRT+10]: extending the concept of production and consumption curves, so-called event count curves are proposed which allow modeling the hierarchical composition and decomposition of event streams.

4.7 Summary

Developers of real-time streaming applications need to continuously quantitatively analyze the timing properties of the system under development throughout the design process. In the context of multiprocessors, this calls for suitable methods to model and analyze the complex timing behavior of applications resulting from the data-and state-dependent behavior of single actors, the distributed execution, and the interference on shared resources. An interesting alternative for this purpose are compositional best-case/worst-case methods, such as Mpa and SymTA/S. Due to their high abstraction level and full coverage of corner cases, these methods are well suited for both system-level performance analysis as well as final system verification.
To make compositional methods applicable to real systems, however, a sufficiently large modeling scope is crucial. Otherwise, many systems might not be analyzable at all or only with a considerable loss of accuracy. Tackling this issue, the contribution of this chapter is a method to analyze actors with complex activation schemes in the framework of \textsc{Mpa}. For this purpose, first a general actor model was introduced that covers actors with multiple inputs and outputs, Boolean activation conditions of actors, and actors consisting of multiple sub-actors. The analysis model for such actors then consists of abstract OR and AND components modeling the activation conditions and greedy processing components modeling the sub-actors whereby the resource sharing of these sub-actors is modeled by a non-preemptive fixed priority scheduling domain. Afterwards, the input–output relations for the abstract OR and AND component as well as for non-preemptive fixed priority scheduling were derived. The derivation of the theoretical results was complemented with several experiments, comparing the proposed method with other analysis methods in several application scenarios. These experiments indicated that the results obtained by \textsc{Mpa} are close to results obtained by alternative methods. It is noteworthy that this is achieved by only using fundamental concepts of \textsc{Mpa} such that the proposed approach can be seamlessly combined with other techniques proposed in the \textsc{Mpa} framework. In particular, the proposed approach is compositional and does not require any kind of global or holistic analysis.

While the analysis presented in this chapter was based on \textsc{Mpa}, the results are more generally applicable. On the one hand, it is conceivable that one can model actors in other analytic performance analysis methods by taking a similar approach. On the other hand, \textsc{Mpa} has been integrated with SymTA/S and timed automata into heterogeneous performance analysis frameworks \cite{KHET07, LPT09}. By extending \textsc{Mpa}, the scope of these heterogeneous analysis frameworks has been extended, as well.
Automated Generation and Calibration of Compositional Performance Analysis Models

Performance analysis is a crucial task in all design phases of real-time multiprocessor streaming applications. Especially when performance-related problems surface late in the design cycle, the consequences are substantial and include increased hardware costs, development cost overruns, time-to-market delays, or even the abandonment of projects. Hence, already during the (early) design space exploration, performance analysis is required to evaluate the performance of different design points besides other quality attributes, such as cost, size, reliability, or safety. Towards the end of the design cycle, performance analysis is required for system verification to check whether a system meets its real-time constraints.

Satisfying this demand, there is a broad spectrum of performance analysis methods that offer various trade-offs between accuracy, scope, setup effort, and analysis time. A performance analyst can, depending on the actual needs, use simple back-of-the-envelope calculations, simulations at different abstraction levels, or analytic methods, for instance. For multiprocessor streaming applications with real-time requirements, compositional best-case/worst-case methods, such as modular performance analysis (Mpa) \cite{CKT03} and symbolic timing analysis for systems (SymTA/S) \cite{HHJ+05} are well suited: they do not rely on complete implementations, have a rather short analysis time, and are exhaustive which makes them suitable for (early) design space
exploration as well as system verification. Still, a performance analyst faces serious challenges when applying compositional best-case/worst-case methods for analyzing multiprocessor streaming applications:

**System Complexity.** Even though an analytic model abstracts from many implementation details, creating a compositional performance analysis model requires a detailed knowledge about the considered system. On the one hand, this includes system-level knowledge about the application, architecture, and mapping. On the other hand, also knowledge about concrete implementation details at lower abstraction levels is required to make sure that the abstractions made conform to the actual system behavior.

**Large Parameter Set.** To characterize a real system, the values for a large number of parameters need to be determined. Fortunately, many parameters are available at design time and merely need to be collected from the application, architecture, and mapping specifications, such as the throughput of an interconnect or the priority of an actor. However, other parameters like the execution time of actors or the run-time overhead of the run-time environment are usually not contained in system specifications and need to be explicitly determined.

**Modeling Effort.** The increasing size of multiprocessors and the steadily improving capabilities of compositional methods are leading to a considerable complexity of analysis models. Creating an analysis model is thus a demanding task that not only requires in-depth knowledge about the specific performance analysis method to be used, but also time and diligence to correctly implement a model.

Consequently, compositional performance analysis for real systems is hardly viable when the analysis model needs to be created by hand. To overcome this problem, the goal is to automate the creation of analysis models. In this chapter, it is shown that a model-driven approach can be used to actually reach this goal for multiprocessor streaming applications. Similar to software synthesis, a compositional performance analysis model can be automatically derived from a system-level specification, assuming that is has been annotated with the required parameters beforehand. This enables the seamless integration of compositional performance analysis into the design flow of multiprocessor streaming applications which, to the best of the author’s knowledge, has not been demonstrated before. Specifically, it is shown how MPA can be integrated into the distributed operation layer (DOL) design flow.
5.1 Problem Statement

The remainder of this chapter is structured as follows: first, the problem to be considered is stated, along with a set of assumptions defining the scope of the proposed approach. In Section 5.2, the proposed approach itself is introduced in general. Afterwards, a prototype implementation of the proposed approach is presented where MPA is integrated into the DoL design flow. Experimental results using this prototype implementation are reported in Section 5.4. Finally, related work is discussed and a summary concludes the chapter. Note that with respect to DoL and MPA, this chapter is based on the concepts and the notation introduced in Chapters 2 and 4.

5.1.1 Assumptions

The following assumptions are made to avoid a mismatch between the analysis model and the actual system behavior, facilitate the characterization of components, and allow for automation, while avoiding to sacrifice performance to a large extent.

AND-Activated Read-Execute-Write Actors. It is assumed that during a firing, actors first read tokens from the input stream(s), process them, and finally write tokens to the output stream(s). Moreover, it is assumed that actors with multiple inputs are fired when tokens are available on all input streams (AND-activation). This assumption is made because the analysis of actors with an arbitrary sequence of read, execute, and write phases, combined with complex activation schemes, has not been addressed by compositional methods, yet.
Absence of Cycles and Back-Pressure. It is assumed that the dataflow process network does not contain cycles (feedback loops) and that the channel buffers are large enough such that an actor never blocks when writing. This assumption is made because compositional performance analysis techniques that have been proposed to handle such scenarios are still limited to the synchronous dataflow model of computation [TS09, JRE05].

Independence of Actors. It is assumed that the best-case/worst-case execution time of an actor firing is independent from other actors executing on the same processor and does not depend on the scheduling. This is assumed in order to allow the best-case/worst-case execution time of actors to be determined independently from each other with a reasonable accuracy.

Note that this assumption conflicts with the use of caches because the cache contents and, consequently, the execution time of an actor's firing, depends on the actors and their scheduling. (To a much smaller extent, this also applies to superscalar and pipelined processors as well as processors with speculative units, such as branch predictors.) Accurately modeling the resulting effects on the performance is still beyond the scope of compositional performance analysis, so that very conservative bounds on the best-case/worst-case execution time would need to be assumed. To reduce the performance penalty by abandoning (hardware-managed) caches, (software-managed) scratchpad memories could be used instead, for instance.

Separation of Communication from Computation. It is assumed that communication is separated from computation, that is, actors can write to and read from channels in constant time (apart from variations due to varying token sizes). In effect, this can be achieved by implementing actors that only write to and read from communication buffers allocated to a local memory that is only accessed by a single actor at any point in time. The end-to-end data transfer between communication buffers allocated to different memories is only triggered by actors but actually executed by separate software components or a direct memory access controller. This also means that the execution time of an actor firing does not depend on the binding of the actors with whom the actor is communicating.

Separation of communication from computation is assumed in order to allow the best-case/worst-case execution time of actors to be determined independently from each other (jointly with the previous assumption). Also, it allows determining tighter bounds on the execution time compared to an actor that needs to carry out communication itself.
5.2. Automated Model Generation and Calibration

Best-case and Worst-case Behavior of Actors Occurs in Simulation. It is assumed that the best-case and worst-case behavior of individual actors can be estimated by functional and cycle-accurate simulation. Specifically, this is assumed regarding workload curves, consumption curves, and production curves. Due to the two previous assumptions, the best-case and worst-case behavior of individual actors is independent from the rest of the system and, consequently, can be estimated by exhaustively simulating an isolated actor, rather than an entire system.

This assumption is made in order to allow for automated calibration of actors by means of simulation. Note that if the simulation cannot be guaranteed to be exhaustive due to complex data-dependent behavior, for instance, the curves obtained by simulation cannot be used for the analysis of hard real-time systems. Still, they can be useful for analyzing systems with soft real-time constraints.

The assumption that the best-case/worst-case behavior of actors occurs in simulation could be dropped when the curves could be obtained otherwise, for instance by (formal) component-level analysis methods or from specifications. As an example, there are formal methods to determine the worst-case execution time \[ WEE^*08 \]. Like other formal methods, however, they suffer from a restricted modeling scope and often cannot be completely automated. Specifications, on the other hand, often (deliberately) do not constrain applications narrowly enough in such a way that useful bounds could be derived. In short, simulation is often the only choice to estimate best-case/worst-case bounds due to the lack of suitable alternatives. For this reason, simulation is also used in \[ MKT04, HMP^*08, SBW09 \], for instance.

5.2 Automated Model Generation and Calibration

In previous chapters, model-driven development (engineering) has been introduced as a methodology that allows automating software synthesis, that is, the platform-specific implementation of an application based on a high-level system model. Similarly, model-driven performance analysis (prediction) refers to the performance analysis of a system based on a high-level system model rather than by simulation or measurement \[ BDMIS04 \]. Following this approach, this chapter is based on the idea that basically the same specification can be used for software synthesis and for generating a compositional performance analysis model, see Fig. 30. The hypothesis is that when both the implementation and the performance analysis model are created by automated transformations, a close match between the two can be maintained throughout the design process, leading to accurate (tight) performance analysis results. The proposed
approach involves two steps that are considered in detail in the following:

**Model Generation.** Model generation refers to the creation of a compositional performance analysis model for a given multiprocessor streaming application.

**Model Calibration.** Model calibration refers to determining the model parameters required to parameterize a compositional performance analysis model.

In this section, a generic overview of this approach is given. Specific details concerning Mpa and Dol will be presented in the following section.

![Figure 30: Approach for automated model generation and calibration.](image)

### 5.2.1 Compositional Performance Analysis Model Generation

During the automated generation of a compositional performance analysis model, two challenges need to be addressed. On the one hand, the relationship between the system specification and the analysis model components and their interaction needs to be established. On the other hand, a compositional performance analysis model is parallel in the sense that it does not specify any particular order in which the model needs to be numerically evaluated. Implementing a compositional performance analysis model in a sequential manner (an Mpa model by a Matlab script, for instance) is thus the second challenge during model generation. Consequently, it takes more than a straightforward source-to-source transformation to automatically create a compositional performance analysis model.
5.2.1.1 Approach

To deal with these challenges, it is proposed to split the model generation into two phases, each addressing one challenge. To fuse these two phases, an intermediate model is proposed that only uses abstractions common to compositional performance analysis methods, see Fig. 31. In the first phase, a system specification is transformed into this intermediate model. In the second phase, a code generator refines the intermediate model using method-specific abstractions and generates the corresponding code. For a description of the concrete implementation of the intermediate model and the two phases in the context of DOL and MPA, the reader is referred to the next section.

![Figure 31: Approach for automated generation of a compositional performance analysis model starting from a synthesizable system specification.](image)

5.2.1.2 Intermediate Model

The proposed intermediate model can be viewed as a directed graph $\mathcal{M} = (V, E)$ which can be annotated with all the information required to analyze the performance of a real-time system. Specifically, the set of vertices $V$ consists of a set of abstract processing components to model computation and communication as well as abstract AND components to model the activation scheme of the actors. The set of edges $E$ represents the event streams modeling the dataflow. The binding of computation and communication components to architectural resources is expressed by partitions of the graph. Information about the resources themselves and the scheduling is conveyed by additional annotations.

The intermediate model at the interface of the two phases allows to easily extend the presented approach: with respect to the creation of the intermediate model, different transformations can be used to transform different kinds of system specifications into the intermediate model. Consequently, as long as a system is within the modeling scope of the intermediate model, only a suitable transformation needs to be
implemented in order to be able to profit from the code generators for specific performance analysis methods. Similarly, different code generators can be used to target different performance analysis methods, such as Mpa or SymTA/S.

5.2.2 Compositional Performance Analysis Model Calibration

In the context of compositional performance analysis and real-time systems, the goal of model calibration is to determine safe bounds on the best-case and worst-case behavior of all system components. At the same time, these bounds should be tight because pessimistic bounds inevitably lead to pessimistic analysis results tantamount to over-dimensioned systems or increased development effort due to code-centric performance fixes.

Conceptually, model calibration is a simple task, so one might easily underestimate the effort it causes. On the one hand, this effort results from the large number of parameters that are needed to characterize an entire system. On the other hand, not all parameters are readily available from specifications, and hence they need to be explicitly determined. No matter whether this is done by measurements of a physical implementation, simulation, formal methods, or benchmarks, this entails the setup of a corresponding environment, the execution of the particular method, and possibly some post-processing. This is already a tedious task for a single design and the problem is even worse for model calibration in the context of design space exploration. Manually performing calibration is thus virtually impossible for real systems.

5.2.2.1 Approach

To automate model calibration, an approach based on simulation is proposed, that is, functional and instruction set simulation are used to determine the parameters of the abstract processing components modeling computation and communication. More specifically, functional simulation is used to parameterize the curves that do not depend on the timing of the application (due to the determinism of dataflow process networks), namely the consumption curves $\kappa$ and the production curves $\pi$ of the abstract processing components modeling actors and the workload curves $\gamma$ for abstract processing components modeling channels. An instruction set simulator, commonly referred to as a virtual platform in the context of multiprocessors, is used to determine the workload curves $\gamma$ for the abstract processing components modeling computation. All the other parameters can be inferred from data sheets, specifications, or one-time measurements, such as resource capabilities, scheduling parameters,
5.2. Automated Model Generation and Calibration

or the context switch overhead of the run-time environment. Note that this approach is based on the assumption that the best-case and worst-case behavior of actors occurs in simulation, as discussed in Section 5.1.1.

Calibration of consumption, production, and workload curves by simulation means that simulation is used to obtain traces that represent the best-case and worst-case behavior of individual actors and channels. These traces are then post-processed to obtain the corresponding curves as described below. This is possible because a curve obtained from a single simulation trace is a bound for an entire set of traces and not just the particular one used for its calibration. In addition, curves obtained using traces from different simulation runs can be easily combined into a single curve by taking the minimum and maximum of, respectively, the lower and upper curves.

For system verification, one can thus use functional simulation and a virtual platform to obtain the required traces for the particular system under consideration. For design space exploration, however, performing independent calibration for each design point using simulation is too slow. Assuming that design space exploration refers to the problem of optimizing the mapping of a fixed dataflow process network onto a fixed architecture, one can observe that the number of curves to calibrate is much smaller than the number of possible mappings because only the workload curves of the actors depend on the mapping. More precisely, due to the assumed independence of the actors and the separation of communication from computation, the workload curve of an actor depends on its binding to a processor (only on heterogeneous architectures with different types of processors, though). All the other mapping-independent curves need to be calibrated just once. A viable strategy for calibration in the context of design space exploration is thus to use functional simulation once to calibrate the timing-independent curves. For calibrating the workload curves of actors, several cycle-accurate simulation runs are carried out to obtain for each actor a trace on each processor where it can be mapped. After completion of these simulation runs, the corresponding curves can be computed and subsequently looked up during design space exploration.

5.2.2.2 Calibration of Workload Curves Using Simulation

The calibration of workload curves (as well as production and consumption curves) by simulation raises the question how a finite simulation trace can be used to parameterize a curve whose domain extends to (positive) infinity. To answer this question, it is assumed that by simulation a finite trace of length \( N \) can be recorded containing the number of cycles required for \( N \) consecutive firings of an actor. By
iteratively summing up the entries, this trace can be converted into a cumulative function $W(e)$, defined as the number of clock cycles required to process $e$ consecutive events. Furthermore, based on the assumption that the best-case and worst-case behavior of actors occurs in simulation, it is assumed that for up to $L$ consecutive firings, $W(e)$ contains safe upper and lower bounds of the workload. This means that for up to $L$ firings, the workload curve $\gamma$ can be obtained by sliding windows of size 1 to $L$ across the cumulative function $W(e)$:

$$\gamma^u(e) = \max_{0 \leq s < N-e} \{W(s + e) - W(s)\} \quad 0 \leq e \leq L \leq N$$

$$\gamma^l(e) = \min_{0 \leq s < N-e} \{W(s + e) - W(s)\} \quad 0 \leq e \leq L \leq N$$

(5.1)

Every interval $e > L$ can now be partitioned into $\lfloor e/L \rfloor$ intervals of length $L$ and a remaining interval of length $e \mod L$. Since $(5.1)$ defines safe upper and lower bounds for any trace, safe bounds for intervals greater than $L$ can be derived by periodically extending $(5.1)$:

$$\gamma(e) = \lfloor e/L \rfloor \cdot \gamma(L) + \gamma(e \mod L) \quad e > L$$

(5.2)

This is also illustrated in Fig. 32. Note that even though the bounds are periodically extended, this does not imply that a stream bounded by these curves needs to be periodic. Clearly, the consumption curves $\kappa$ and production curves $\pi$ can be calibrated following the same approach.

**Figure 32:** Construction of workload curve (right) from a finite trace of accumulated workloads (left).

### 5.3 MPA Model Generation and Calibration in DOL

In the previous section, a general approach for the automated generation and calibration of compositional performance analysis models was
5.3. MPA Model Generation and Calibration in DOL

introduced. In this section, details of a prototype implementation of this approach in the context of DOL using MPA are presented.

5.3.1 MPA Model Generation

In the proposed approach, the first phase in performance analysis model generation is the transformation of a system specification consisting of an application, architecture, and mapping specification into an intermediate analysis model. In this prototype implementation, this model is stored in an XML-based format whose scope includes:

- Event Sources: Event sources abstract the timing behavior of external inputs. The periodic-with-jitter/burst model as well as arrival curves can be used to specify the timing behavior.

- Abstract Processing Components: Abstract processing components model the computation and communication in a multiprocessor streaming application and are characterized by their best-case/worst-case execution time or by workload curves. Actors with multiple inputs are modeled using abstract AND components.

- Architectural Resources: Computation and communication resources as well as the binding of abstract processing components to these resources are modeled by defining scheduling domains that group together the processing components. The capability (availability) of a resource can be described by its clock frequency or the bounded delay model.

- Scheduling Policies: Supported scheduling policies are preemptive and non-preemptive fixed priority scheduling, time division multiple access, first-come first-serve, and earliest deadline first. In addition, hierarchical scheduling domains can be expressed.

Due to the close semantic match between a dataflow process network and the (intermediate) compositional performance analysis model as well as the assumptions stated in Section 5.1.1 the transformation of a DOL specification into the intermediate model involves only a few steps:

1 Abstract processing components are instantiated, as follows: actors are modeled by abstract processing components and an additional abstract AND component when the actor has multiple inputs. Software channels for inter-processor communication are modeled as abstract processing components as well. Software channels connecting actors on the same processor, however, are not explicitly modeled because they do not consume processing resources. (The copying of data to and from the channel buffer is attributed to the corresponding actors.)
2 Abstract processing components are connected by event streams according to the dataflow in the process network.

3 Finally, information from the architecture and mapping specifications is used to accordingly partition the graph and add other annotations that are relevant for performance analysis.

In a second step, the directed graph $M = (V, E)$ that has been generated this way needs to be translated into a model that is specific to the chosen performance analysis method. In the case of DoS, the concrete goal is to implement the intermediate model as a Matlab script based on the Matlab MPa toolbox \cite{WT06c}. This can be achieved as follows:

1 The intermediate model is first expressed using the MPa specific abstractions: event streams are replaced by arrival curves, abstract processing components by greedy processing components (GPCs), and the mapping information is expressed using service curves. The result can again be seen as a directed graph whose vertices are GPCs and whose edges are arrival and service curves.

2 To implement the graph as a sequential declarative specification (as a Matlab MPa script in the particular case of MPa) the causal dependencies of arrival and service curves need to be resolved. Basically, this can be done by performing a topological sort on the graph, that is, by linearly ordering the vertices such that each vertex appears before all vertices to which it has an outbound edge. Afterwards, the corresponding statements can be written to a file in the order computed.

3 Topological sorting is only possible for an acyclic directed graph, however. Unfortunately, an MPa model may contain cyclic dependencies even though the dataflow process network does not contain any cycles, as stated in the assumptions. To resolve such cyclic dependencies, fixed point iterations are used which has been formally shown to be a correct approach in \cite{JPTY08}.

An example is shown in Fig. 33 where the arrival curve and the service curve connecting the GPCs A2 and A3 form a cycle. To resolve this cycle, as a starting point it is assumed that the input service curve of A2 is $\beta_{DSP}$. Afterwards, the input–output relations of A2 and A3 are iteratively evaluated until the fixed point (convergence) is reached. Note that the choice of the starting point of the fixed point iteration potentially has a large influence on the final fixed point and even whether a fixed point can be reached at all. In this sense, the pragmatic choice of $\beta_{DSP}$ as a starting point might not be optimal. The reader is referred to \cite{JPTY08} for further details concerning this aspect.
5.3. **MMPA Model Generation and Calibration in DOL**

**Figure 33:** DOL specification (left) and the according MMPA model (right). Only the channel C1 is modeled in the MMPA model because unlike C2, it occupies the bus. On the DSP, fixed priority scheduling is employed where the actor A3 has a higher priority than the actor A2, leading to a cyclic dependency in the MMPA model.

5.3.2 **MMPA Model Calibration**

To calibrate the workload, production, and consumption curves of GPCs, functional simulation and timed simulation on a virtual platform are used. In this context, following a model-driven development approach as in DOL has the advantage that the existence of a formal programming model greatly facilitates automating this task. Specifically, simulation log files serving as the basis for calibration can be obtained by monitoring the API primitives `fire()`, `read()`, and `write()` introduced in the programming model. After simulation, the obtained log files are post-processed to compute the corresponding curves which are finally back-annotated to the system specification for reference during the MMPA model generation.

In DOL, functional simulation and timed simulation are used as follows:

- **Functional Simulation:** For the consumption and production curves of actors, how often `read()` and `write()` are invoked is monitored. For the workload curve of channels, how many bytes are written in each invocation of `write()` is monitored.

- **Timed Simulation:** For the workload curves of actors, when actors start and finish their `fire()` routine is monitored.

Technically, there are different possibilities for how the monitoring can be realized. One possibility is to instrument the application code. In functional simulation, for example, the application-independent `read()` and `write()` routines as well as the scheduler can be instrumented to produce a log file containing the required data. In timed simulation, however, this approach is less applicable because the instrumentation changes the timing behavior of the simulation itself. Still, sometimes it is possible to insert special instructions for instrumentation that invoke
callback routines (hooks) on a virtual platform which can be used for logging the simulation. Due to the low overhead of these instructions, they remain in the source code even for the final implementation, avoiding any difference to the application code used for calibration.

Another possibility is to take advantage of a debugger for monitoring. Basically, the approach is to first set breakpoints on `fire()`, `read()`, and `write()`. Whenever a breakpoint is hit, a corresponding message is logged to a file. This approach is illustrated in Fig. 34 for the virtual platform of the Atmel Dspis 940 which features a tool command language (Tcl) interface for batch execution: based on a DoL system specification and the (automatically generated) binary, a Tcl script is generated containing the addresses of breakpoints as well as the hooks that are invoked whenever a breakpoint is hit. To create a log file, the virtual platform is then simply executed under the control of this script.

Note that for timed simulation, this log file cannot only be used to extract workload curves but also for creating a visualization of the simulation trace, as shown in Fig. 34. As the visual inspection of a simulation trace is rather intuitive, this visualization can serve as a valuable tool for average-case performance analysis.

Following this approach, multiple simulation runs are executed to exhibit the best-case and worst-case behavior of individual actors, either by using different bindings of actors to processors or using different input streams covering data-dependent behavior of actors. Once more note that this approach is in most cases not adequate for the calibration of hard real-time systems because it usually cannot be guaranteed that simulation exhaustively covers all corner cases of all components.

![Diagram of MPA model calibration in DoL](http://gtkwave.sourceforge.net)

**Figure 34:** MPA model calibration in DoL. The simulation trace is stored in a value-change dump file (IEEE standard 1364-1995) and visualized using GTKWAVE. A magnified view of the simulation trace is shown in the Appendix A.3.
5.4 Experimental Results

In this section, a prototype implementation of DoL, extended by the approach for automated generation and calibration of Mpa models, is used to demonstrate the viability of the proposed approach by analyzing three multiprocessor streaming applications: a producer–consumer application, a motion JPEG decoder application, and a wave field synthesis application. All applications run on top of a multiprocessor ARM platform (Mparm) [BBB+05].

5.4.1 Hardware/Software Multiprocessor Platform

Mparm is a typical example for the class of multiprocessors considered in this thesis, see Section 2.1. That the proposed approach is also applicable to other multiprocessors is shown in [HKH+09], for instance, where the same approach is used to analyze multiprocessor streaming applications running on the Atmel Dipsis 940.

Mparm is a scalable multiprocessor that consists of fully programmable ARM7 based subsystems (tiles) and a shared bus, as shown in Fig. 35. Mparm has a distributed memory architecture, that is, the ARM subsystems communicate via message passing whereby the communication buffers are allocated to the scratchpad memories.

Concerning the assumptions made in Section 5.1.1, Mparm has the following properties: to ensure the independence of actors executing on the same tile, caches are disabled and processors read instructions and data directly from local (scratchpad) memories. To ensure the separation of communication from computation, the available direct memory access (DMA) controllers are used. Located at the interface between a tile and

![Figure 35: Block diagram of the Mparm architecture. The blocks labeled with “M” and “S” denote master and slave ports on the bus.](image-url)
the bus, a DMA controller can autonomously handle the message queue communication between actors without requiring processor resources.

The real-time executive for multiprocessor systems (RTEMS) [RTE10] is used as the real-time operating system which provides a mechanism for quasi-parallelism and FIFO communication. Specifically, the actors of a dataflow process network application that are bound to each tile are implemented as RTEMS tasks and the channels as RTEMS message queues. Although RTEMS supports time-triggered as well as event-triggered scheduling, only preemptive fixed priority scheduling is used in what follows.

5.4.2 Experimental Setup

A virtual platform modeling MpARM is used to carry out the experiments. In this virtual platform, the ARM cores are modeled as cycle-accurate instruction set simulators whereas transaction-level modeling is used for the bus and the attached bus masters and slaves. The role of the platform is twofold: on the one hand, it is used for the calibration of MpA models, on the other hand, the bounds obtained by MpA are compared with measurements obtained on the virtual platform to argue about the accuracy (tightness) of the bounds. Note that an accurate, quantitative evaluation of the tightness of the bounds would require an exhaustive simulation which is too time-consuming for the applications considered. Even without exhaustive simulation, however, general trends can be identified, and for this reason comparisons between results from non-exhaustive simulations and best-case/worst-case analyses are frequently used to obtain information about the accuracy of methods, see [KHET07, HMP+08, SBW09], for instance.

The generation and calibration of MpA models follows the steps described in the previous section. For this purpose, the Dol design flow, as introduced in Chapter 2, has been extended by three packages implementing the transformation of a Dol specification into the intermediate model, the transformation of the intermediate model into a Matlab MpA script, and the tool-chain for calibration. Specific to MpARM and the applications considered are the following aspects:

- The run-time overhead of RTEMS is subsumed in the costs for context switching. Considering that an actor might preempt another one when firing and releasing the resource after completion, this overhead is modeled by computing an adjusted workload curve $\gamma_{\text{new}}$ for all actors based on the original workload curve $\gamma$ and the worst-case context
5.4. Experimental Results

switch time $T_{\text{context}}$ as follows:

$$
\gamma'_{\text{new}}(e) = \gamma''(e) + 2 \cdot e \cdot T_{\text{context}}
$$

$$
\gamma''_{\text{new}}(e) = \gamma'(e)
$$

- Communication on the bus is not explicitly modeled because its effect on the timing is negligible in the applications considered. On the one hand, this is due to the low communication-to-computation ratio of the applications. On the other hand, this is due to the high bandwidth of the bus which is dimensioned for a system integrating more than 20 ARM tiles of which not more than 3 are used in the applications.

- The actors in these applications do not exhibit variable token consumption or production. In each firing, one token is read from each input channel and one token is written to each output channel.

5.4.3 Applications

Three applications are considered, namely a producer–consumer application, a motion JPEG decoder application, and a wave field synthesis application. Two configurations are considered for all applications. The difference between these configurations is the priority of the actors. In the first configuration, an actor located earlier in a chain of actors has a higher priority than an actor located later in the chain. In the second configuration, the priority assignment is reversed, that is, an actor located earlier in a chain has lower priority than an actor located later in the chain.

Note that in Tables 9–14, the time intervals are given in units of 1000 clock cycles. (Operating at a frequency of 100 MHz, 1000 clock cycles amount to 10 µs in Mparm.) Backlogs are specified in the number of tokens and lower numerical values refer to higher priorities.

5.4.3.1 Producer–Consumer

The producer–consumer application consists of three actors mapped onto two processors, as shown in the corresponding Mpa model in Fig. 36. The producer generates a stream of floating point numbers which pass through the worker and are finally printed by the consumer. The producer is activated at a period of $0.2 \cdot 10^6$ clock cycles with a jitter of $10^6$ clock cycles.

Note that this application is also used as an example in the Appendix. Among others, the intermediate model and the corresponding Matlab Mpa script of this application are shown in Section A.3.
Chapter 5. Generation and Calibration of Performance Analysis Models

5.4.3.2 Motion JPEG Decoder

Motion JPEG (MJPEG) \cite{Wal92} is a video codec in which each video frame is separately compressed as a JPEG image. Similar to the MJPEG decoder introduced in Chapter 3, the MJPEG decoder application considered here leverages the data parallelism available in MJPEG decoding. The processing takes place in five subsequent actors: (1) The video stream is first split up into individual frames which (2) are variable length decoded and split up into individual macroblocks. (3) The individual macroblocks are decoded. (4) The macroblocks are stitched together into frames and finally (5) into a stream. The decoder is mapped onto three processors, resulting in the \textit{M}pA model depicted in Fig. 37. The split stream actor is activated at a period of $0.2 \cdot 10^6$ clock cycles with a jitter of $10^6$ clock cycles.

Table 9: Comparison of values observed in simulation and computed worst-case bounds (in parentheses) for the producer–consumer application.

<table>
<thead>
<tr>
<th>ACTOR</th>
<th>EX. TIME</th>
<th>PROC.</th>
<th>PRIO.</th>
<th>BACKLOG</th>
<th>DELAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCER</td>
<td>[56, 63]</td>
<td>1</td>
<td>1</td>
<td>6 ≤ 6</td>
<td>305 ≤ 331</td>
</tr>
<tr>
<td>WORKER</td>
<td>[56, 59]</td>
<td>2</td>
<td>1</td>
<td>2 ≤ 2</td>
<td>63 ≤ 75</td>
</tr>
<tr>
<td>CONSUMER</td>
<td>[52, 59]</td>
<td>1</td>
<td>2</td>
<td>8 ≤ 9</td>
<td>604 ≤ 703</td>
</tr>
<tr>
<td>PRODUCER</td>
<td>[56, 63]</td>
<td>1</td>
<td>2</td>
<td>6 ≤ 10</td>
<td>632 ≤ 1225</td>
</tr>
<tr>
<td>WORKER</td>
<td>[56, 59]</td>
<td>2</td>
<td>1</td>
<td>1 ≤ 2</td>
<td>61 ≤ 70</td>
</tr>
<tr>
<td>CONSUMER</td>
<td>[52, 59]</td>
<td>1</td>
<td>1</td>
<td>1 ≤ 2</td>
<td>59 ≤ 61</td>
</tr>
</tbody>
</table>

Figure 36: \textit{M}pA model of producer–consumer application (configuration 1).
5.4. Experimental Results

### 5.4.3.3 Wave Field Synthesis

By using a loudspeaker array and audio beamforming techniques, wave field synthesis allows to reproduce an acoustic sound field whose perceived origin is not restricted to the position of the physical loudspeaker boxes \[BdVV93\]. The wave field synthesis application considered here renders a sound source using 16 loudspeakers and consists of five actors mapped to two processors, as shown in the corresponding \(\text{M\textsc{pa}}\) model in Fig. 38. The source actor receives a mono audio signal and the control actor receives the beamforming coefficients, both at a period of \(0.8 \cdot 10^6\) clock cycles with a jitter of \(0.8 \cdot 10^6\) clock cycles. The signal processing is performed in the compute actors where each of

---

**Figure 37:** \(\text{M\textsc{pa}}\) model of MJPEG decoder application (configuration 2).

**Table 10:** Comparison of values observed in simulation and computed worst-case bounds (in parentheses) for the MJPEG decoder application.

<table>
<thead>
<tr>
<th>ACTOR</th>
<th>EX. TIME</th>
<th>PROC.</th>
<th>PRIO.</th>
<th>BACKLOG</th>
<th>DELAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLIT STREAM</td>
<td>[10, 15]</td>
<td>1</td>
<td>1</td>
<td>3 ((\leq 5))</td>
<td>24 ((\leq 50))</td>
</tr>
<tr>
<td>SPLIT FRAME</td>
<td>[53, 59]</td>
<td>2</td>
<td>1</td>
<td>5 ((\leq 6))</td>
<td>239 ((\leq 277))</td>
</tr>
<tr>
<td>DECODE</td>
<td>[133, 135]</td>
<td>3</td>
<td>1</td>
<td>5 ((\leq 6))</td>
<td>649 ((\leq 706))</td>
</tr>
<tr>
<td>MERGE FRAME</td>
<td>[33, 39]</td>
<td>2</td>
<td>2</td>
<td>3 ((\leq 4))</td>
<td>352 ((\leq 481))</td>
</tr>
<tr>
<td>MERGE STREAM</td>
<td>[12, 18]</td>
<td>1</td>
<td>2</td>
<td>1 ((\leq 3))</td>
<td>35 ((\leq 97))</td>
</tr>
</tbody>
</table>

---

5.4.3.3 Wave Field Synthesis

By using a loudspeaker array and audio beamforming techniques, wave field synthesis allows to reproduce an acoustic sound field whose perceived origin is not restricted to the position of the physical loudspeaker boxes \[BdVV93\]. The wave field synthesis application considered here renders a sound source using 16 loudspeakers and consists of five actors mapped to two processors, as shown in the corresponding \(\text{M\textsc{pa}}\) model in Fig. 38. The source actor receives a mono audio signal and the control actor receives the beamforming coefficients, both at a period of \(0.8 \cdot 10^6\) clock cycles with a jitter of \(0.8 \cdot 10^6\) clock cycles. The signal processing is performed in the compute actors where each of
the two actors synthesizes the signals for eight channels. The computed audio signals are communicated to the speaker actor which interfaces with the analog part of the system.

![Diagram](image)

**Figure 38:** MPA model of WFS application (configuration 1).

<table>
<thead>
<tr>
<th>ACTOR</th>
<th>EX. TIME</th>
<th>PROC.</th>
<th>PRIO.</th>
<th>BACKLOG</th>
<th>DELAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL</td>
<td>[49, 54]</td>
<td>1</td>
<td>1</td>
<td>1 (&lt; 2)</td>
<td>50 (&lt; 86)</td>
</tr>
<tr>
<td>SOURCE</td>
<td>[44, 54]</td>
<td>1</td>
<td>2</td>
<td>2 (&lt; 2)</td>
<td>115 (&lt; 190)</td>
</tr>
<tr>
<td>COMPUTE A</td>
<td>[321, 348]</td>
<td>2</td>
<td>1</td>
<td>4 (&lt; 14)</td>
<td>495 (&lt; 3298)</td>
</tr>
<tr>
<td>COMPUTE B</td>
<td>[321, 348]</td>
<td>2</td>
<td>2</td>
<td>7 (&lt; 15)</td>
<td>1839 (&lt; 4778)</td>
</tr>
<tr>
<td>SPEAKER</td>
<td>[167, 179]</td>
<td>1</td>
<td>3</td>
<td>3 (&lt; 14)</td>
<td>1443 (&lt; 4869)</td>
</tr>
<tr>
<td>CONTROL</td>
<td>[49, 54]</td>
<td>1</td>
<td>3</td>
<td>2 (&lt; 2)</td>
<td>306 (&lt; 551)</td>
</tr>
<tr>
<td>SOURCE</td>
<td>[44, 54]</td>
<td>1</td>
<td>2</td>
<td>2 (&lt; 2)</td>
<td>227 (&lt; 269)</td>
</tr>
<tr>
<td>COMPUTE A</td>
<td>[321, 348]</td>
<td>2</td>
<td>2</td>
<td>9 (&lt; 19)</td>
<td>2165 (&lt; 6264)</td>
</tr>
<tr>
<td>COMPUTE B</td>
<td>[321, 348]</td>
<td>2</td>
<td>1</td>
<td>4 (&lt; 18)</td>
<td>711 (&lt; 4241)</td>
</tr>
<tr>
<td>SPEAKER</td>
<td>[167, 179]</td>
<td>1</td>
<td>1</td>
<td>4 (&lt; 15)</td>
<td>1866 (&lt; 5691)</td>
</tr>
</tbody>
</table>

Finally, the following tables list the workload curves $\gamma^u(e)$, $\gamma^l(e)$ for the actors in the three applications. The values of the workload curves have been determined by simulation up to $e \leq L = 15$. For $e > L$, the workload curves are periodically extended, as described in Section 5.2.2.2.
5.4. Experimental Results

5.4.4 Discussion

In this chapter, it has been argued that performance analysis is, potentially, among the most time-consuming tasks in the design of real-time multiprocessor streaming applications. This calls for design automation, and the hypothesis was posed that by automatically generating a performance analysis model from the same specification that is used for software synthesis, accurate performance analysis results can be obtained.
Furthermore, it was proposed that for design space exploration model calibration is performed only once in order that the exploration itself can be carried out efficiently, because the required parameters for different design points will merely need to be looked up. In the following, these two aspects will be discussed.

5.4.4.1 Accuracy

Out of the 52 bounds listed in Table 9, 10, and 11, 22 are within 20% of the observed values and another 13 are within 100% of the observed values. In the remaining 17 cases, the observed values and bounds are further apart and the maximal mismatch occurs for the delay of the compute B actor in configuration 2 of the wave field synthesis application (an observed delay of $0.711 \cdot 10^6$ cycles vs. a computed delay bound of $4.241 \cdot 10^6$ cycles). A differentiated view is necessary to interpret this result:

On the one hand, a large fraction of these bounds is within 20% of the observed values: this can be considered as a satisfactory accuracy for best-case/worst-case compositional performance analysis. Deviations in this range are frequently observed, see [PWT+09], for instance, where results for five benchmark applications are reported.

On the other hand, there is a deviation of more than a factor of 2 between the observed values and the bounds for a significant fraction of the bounds. This can be attributed to the following factors.

- The observed values have been obtained by (non-exhaustive) simulation, so it is unclear which of the observed values are close to the actual worst-case and which not.

- Resolving cyclic dependencies by means of fixed point iteration causes inaccuracies. In Perathoner et al. [PWT+09], the end-to-end delay of a simple system with cyclic dependencies is overestimated by a factor of almost five when using MPAL or SymTA/S, for instance.

- In all systems, the first and the last actor in the processing chain are bound to the same processor. This gives rise to timing correlations that are not taken into consideration in the prototype implementation used for the experiments: the generated MPAL model implies that the worst-case workload of all actors belonging to a scheduling domain might be triggered at exactly the same moment whereas timing correlations may actually prevent this from happening in the real system.

- Similarly, the timing correlations between multiple streams activating a single actor are not taken into account in the prototype implementation.
used for the experiments which leads to an overestimation of the backlog and delay bounds for AND-activated tasks.

Not surprisingly, the maximum mismatch occurs for an actor where cyclic dependencies and timing correlations are present. On the other hand, for the producer–consumer and MJPEG decoder applications, 18 out of 32 bounds lie within 20% of the observed values and only in three cases are the bounds not within 100% of the observed values.

Consequently, it can be concluded that the accuracy of the bounds deteriorates when the compositional performance analysis model does not accurately model effects occurring in the real system. To further increase the accuracy of such bounds, primarily the analysis of systems with timing correlations and cyclic dependencies would need to be improved. This could be achieved by extending the developed prototype implementation by suitable techniques where [HTSD07] and [PWT+09] could serve as a starting point. This does not require any conceptual change of the proposed approach, however.

### 5.4.4.2 Generation and Calibration Effort

In the following, an intuition about the duration of automated generation and calibration of compositional performance analysis models is given. For this purpose, the durations for performing different design steps have been measured for the three applications considered, see Table 15.

<table>
<thead>
<tr>
<th>DESIGN STEP</th>
<th>PC</th>
<th>MJPEG</th>
<th>WFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>model calibration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>functional simulation generation</td>
<td>21 s</td>
<td>22 s</td>
<td>23 s</td>
</tr>
<tr>
<td>functional simulation</td>
<td>1 s</td>
<td>1 s</td>
<td>1 s</td>
</tr>
<tr>
<td>synthesis (generation of binary)</td>
<td>7 s</td>
<td>7 s</td>
<td>7 s</td>
</tr>
<tr>
<td>simulation on virtual platform</td>
<td>90 s</td>
<td>118 s</td>
<td>345 s</td>
</tr>
<tr>
<td>log-file analysis</td>
<td>10 s</td>
<td>10 s</td>
<td>18 s</td>
</tr>
<tr>
<td>generation of intermediate model and MPA script</td>
<td>&lt; 2 s</td>
<td>&lt; 2 s</td>
<td>&lt; 2 s</td>
</tr>
<tr>
<td>performance evaluation in Matlab</td>
<td>&lt; 1 s</td>
<td>&lt; 1 s</td>
<td>&lt; 1 s</td>
</tr>
</tbody>
</table>
Based on the reported durations, the following conclusions can be drawn:

- Even though the calibration is completely automated, it takes a considerable time. Thus, manual calibration at the same level of detail would be a very time-consuming endeavor. This shows that automated model calibration is essential when compositional performance analysis is applied to analyzing real systems.

- Once the calibration data are available, model generation and performance analysis itself are rather fast. Especially compared to simulation on a virtual platform, evaluating a system’s performance by compositional performance analysis is faster by several orders of magnitude. This justifies the integration of compositional performance analysis into the design cycle, not only for the purpose of final system verification but also for design space exploration.

Finally, Table 16 shows the number of source code lines of the automatically generated Matlab M scripts. Once again, the numbers indicate that a manual approach would very soon come to a limit if more than a few different systems needed to be modeled.

### Table 16: Code size of Matlab M scripts for the applications considered here in terms of the number of source code lines. The two columns refer to the two different priority configurations. The code for implementing the fixed-point iteration leads to an increased line count (right column).

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>WITHOUT CYCLIC DEP.</th>
<th>WITH CYCLIC DEP.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCER–CONSUMER</td>
<td>72</td>
<td>94</td>
</tr>
<tr>
<td>MJPEG DECODER</td>
<td>97</td>
<td>141</td>
</tr>
<tr>
<td>WAVE FIELD SYNTHESIS</td>
<td>120</td>
<td>142</td>
</tr>
</tbody>
</table>

### 5.5 Related Work

Performance is a pervasive issue in software development which affects all layers in the software stack, from the application itself to the hardware abstraction layer. Consequently, research has often addressed the integration of performance analysis into the software development process. One research direction sometimes referred to as software performance engineering [WFP07] deals with systematic, quantitative approaches to construct software systems that meet performance
5.5. Related Work

requirements. Model-based performance analysis (prediction) can be seen as one possible approach in this direction.

Extensive surveys about research projects in software performance engineering are presented in [Koz10, BDMIS04]. These surveys reveal that one major application of model-based performance analysis is large-scale, component-based, distributed systems communicating via (best-effort) networks, such as web and database systems. The approach is that component developers provide performance models that convey resource demands (processors, memories), service demands (threads, synchronization, communication), or even more fine-grained information (control flow, component state) of software components. Besides custom specification languages, the unified modeling language (UML) [UML10] and its extensions for modeling performance attributes are frequently used for this purpose, namely UML-SPT (UML profile for schedulability, performance, and time specification) [SPT05] and its successor UML-MARTE (UML profile for modeling and analysis of real-time and embedded systems) [MAR09]. A concrete mapping (deployment) of an application onto an architecture is subsequently expressed in an abstract performance model, such as (extended or layered) queuing networks, (stochastic) Petri nets, or different kinds of stochastic process algebras.

The approach from this research direction that is most closely related to the one presented in this chapter is PACC (predictable assembly from certifiable components) [MM08]: PACC is a project focusing on the design of control-oriented real-time applications. Similar to Dol, PACC supports both software synthesis and performance analysis from a single system model. It is also based on a restrictive model of computation, uses an intermediate format for expressing performance analysis models, and features an automated transformation of these models into implementations for an existing performance analysis framework: specifically, systems are expressed in a custom, domain-specific language based on UML statecharts, the so-called construction and composition language. For performance analysis, systems expressed in that language are first transformed into a so-called intermediate constructive model that contains all the information relevant to performance analysis. In a second step, this intermediate constructive model can then be transformed into a MAST (modeling and analysis suite for real-time applications) [HGCMM01] model for rate-monotonic analysis. Further parallels between Dol and PACC can be identified when looking at an industrial case study [HMP*08] where PACC has been used for the performance analysis of a monitoring and control system. In that case study, the best-case and worst-case execution times of software components are determined by simulation.
Furthermore, the predicted worst-case latencies are between 1.6 and 2.6 times larger than the observed worst-case latencies, which is similar to the results reported in this chapter. Hence, the PACC project seems to confirm the viability of the overall approach presented in this chapter, its division into different phases, and the accuracy of the obtained bounds.

A second research direction in which performance analysis plays a central role is the hardware/software codesign of distributed embedded systems. Many frameworks following a model-driven development approach have been proposed for this purpose, see [DSVP06, GHP09, Gri04] for comprehensive surveys. One can observe that the most widely used approach for performance analysis in these frameworks is simulation. Perhaps the reason is that cycle- or instruction-accurate virtual platforms are typically well integrated into the design flow as a tool for software development, testing and debugging. Consequently, it is rather convenient to leverage them for performance analysis, as well. The relatively slow simulation of multiprocessors is offset by a low setup effort, assuming that the virtual platform already exists and implementations can be generated by automated software synthesis.

For early design space exploration, however, a short analysis time is vital: for this reason, simulation using virtual platforms is not a viable option. Still, simulation can be used for early design space exploration by raising the abstraction level, which can reduce the simulation time by several orders of magnitude. One appropriate simulation technique for multiprocessor streaming applications is trace-based simulation. In trace-based simulation, the behavior of a dataflow process network is abstracted by an architecture- and mapping-independent (untimed) trace of (computation and communication) events. The timing behavior of an application when mapped onto an architecture is then estimated by unfolding this trace in time by “executing” it on virtual processing resources modeling the real architecture. Using discrete-event simulation for this purpose, the average-case timing behavior of multiprocessor streaming applications can be quickly estimated with high accuracy. This approach has been explored in Dol [HBLH09] and Artemis [Pim08], for instance, where errors of less than 3% compared to instruction set simulation are achieved. For calibration, that is, for determining the resource requirements of computation and communication events, instruction set simulation is used in both design flows.

Being non-exhaustive, simulation cannot be used to determine hard bounds on timing properties, however, which limits the scope of design flows that are solely based on simulation for performance analysis. To extend the scope of the Dol. design flow to the design of real-time multiprocessor streaming applications, compositional performance
5.6. Summary

Whenever multiprocessor streaming applications need to fulfill (real-time) timing requirements, performance analysis becomes an important task in the design cycle. In this case, performance analysis needs to be repeatedly carried out from early design space exploration to final system verification, and support for design automation is thus highly desirable. On the one hand, performance analysis requires deep knowledge about the system to model as well as the analysis method used. On the other hand, the effort for generating and calibrating a concrete instance of a performance analysis model is considerable, even if this knowledge exists. In particular, this applies to performance analysis by means of compositional best-case/worst-case methods because the employed abstractions are not widely used.

Tackling these challenges, the work presented in this chapter is the first attempt to integrate compositional performance analysis into a design flow for model-driven development of multiprocessor streaming applications. This way, the exhaustive and scalable performance analysis provided by compositional methods is paired with a low setup effort. After introducing the general approach, specific details of a prototype implementation in the context of the DOL design flow were presented. The experimental results using this implementation confirmed the viability of the approach: they showed that the analysis results for a calibrated model match reasonably well with the observations of the real system implementation. At the same time, these experiments revealed that ignoring timing correlations, for instance, leads to overly pessimistic results and suitable techniques would need to be incorporated into the approach to widen its scope.

In conclusion, this chapter confirmed the merits of model-driven development: thanks to a restricted system specification, performance analysis has been integrated into the design flow. This integration can also be considered as a new application for compositional performance analysis itself because previously it has been mainly applied during early design space exploration [KTZ05, WTVL06, RHE07]. In that case, the analysis models are generated separately from the system specification used for system synthesis because the analysis model used for exploration contains different information than required for a synthesizable specification. Hence, the automated calibration of compositional performance analysis models has only been addressed to a very limited extent because up until now, estimation rather than calibration has been used to determine the model parameters during early design space exploration.
analysis could be completely automated without suffering from a significant drawback compared to a manual approach. Performance analysts can thus concentrate on the interpretation of the results and their consequences for the design, rather than spending time on obtaining the results.
Conclusions

The main conjecture of this thesis is that a design flow based on the dataflow process network model of computation can facilitate the development of multiprocessor streaming applications by automating several design steps. To support this conjecture, it is demonstrated that software synthesis and performance analysis can indeed be automated, relieving designers of the need to carry out these steps manually. In this chapter, the contributions of this thesis are summarized, conclusions are drawn, and future directions are outlined.

6.1 Summary of the Contributions

Lightweight Multiprocessor Process Network Run-Time Environment. A run-time environment for dataflow process networks is proposed that mitigates one drawback of the dataflow model, namely that the communication between actors and their orchestration can considerably limit the speed-up achievable by parallel execution. The key feature of the run-time environment is that so-called protothreads [DSVA06] are used to implement the quasi-parallelism required for executing multiple actors on a single processor. In addition, windowed FIFOs are used instead of standard FIFOs for efficient inter-process communication. The run-time overhead and memory footprint of this run-time environment is smaller compared to existing ones, and it can be easily ported to different multiprocessors. In particular, the run-time environment has been ported to the Sony/Toshiba/IBM Cell Broadband Engine where experiments have been performed to show its efficiency.
**Mpa of Actors with Complex Activation Schemes.** The scope of modular performance analysis (Mpa) is extended to actors with multiple inputs and outputs that exhibit complex activation schemes. First, a corresponding process model is introduced that extends the well-known AND/OR-activation scheme of actors to more general activation schemes. For actors adhering to this model, subsequently a best-case/worst-case analysis method is proposed which combines a newly devised method for analyzing non-preemptive fixed priority scheduling domains with existing methods from Mpa.

**Automated Generation and Calibration of Mpa Models.** A method for the automated creation of Mpa models is presented. This method consists of two steps, namely model generation — the generation of an Mpa model for a given system — and model calibration — the acquisition of the required model parameters. Following a top-down approach, the Mpa model is derived from the specification that is also used for software synthesis and, thus, no additional user input is required. Insight into the potential and limitations of this approach is provided by experiments on a multiprocessor ARM platform [BBB+05].

**Integration into the Distributed Operation Layer.** The contributions discussed above are integrated into the distributed operation layer (Dol) design flow. The Dol is a design flow to support the development of multiprocessor streaming applications based on the dataflow model of computation. By integrating the proposed techniques into the Dol, its scope is extended towards the development of real-time multiprocessor streaming applications.

### 6.2 Conclusions

The architectural characteristics of multiprocessors, such as heterogeneity, explicitly managed memory hierarchies, or memory and communication bandwidth limitations, make the design and performance analysis of multiprocessor applications a difficult task. In the long term, designing and analyzing entire multiprocessor applications at device-level is not realistic. Therefore, raising the abstraction level and, consequently, the degree of automation is necessary. One way of raising the abstraction level is to follow a model-driven development approach, such as the one proposed in this thesis. Specifically, one can draw the following conclusions considering the two main contributions of this thesis, software synthesis and performance analysis for multiprocessor streaming applications.
Concerning software synthesis, one could conclude that a model-driven development approach based on dataflow process networks is well suited for designing multiprocessor streaming applications. Using the proposed run-time environment based on protothreads and windowed FIFOs, the run-time overhead for executing a process network can be kept rather low, so even a hand-written implementation can barely outperform the automatically generated one. (Of course, this assumes that the application can be efficiently modeled as a process network.) Also, except for the windowed FIFO implementation, the run-time environment can be written in pure ANSI C, that is, without any platform-specific assembly code. Thus, it can easily be ported to different platforms. The biggest shortcoming of the run-time environment is that it only has a cooperative scheduler. Therefore, its applicability to hard real-time systems is limited where usually fully preemptive scheduling policies are preferred, such as earliest deadline first scheduling or time-triggered policies.

Concerning performance analysis, the conclusions are two-sided. On the one hand, it has been shown that the generation of a modular performance analysis model for real-time multiprocessor streaming applications can indeed be completely automated. Hence, a system developer can concentrate on interpreting the obtained results and improving the design instead of spending time on the tedious process of analysis model generation and calibration. On the other hand, applying best-case/worst-case performance analysis methods to real multiprocessor systems exposed problems in the analysis that are less accentuated in the early design stages onto which the majority of case studies in the related literature is focused. First, some powerful techniques proposed in the literature are inherently difficult to automate because the functional specification and simulation do not convey the required information. To be able to faithfully model and analyze effects, such as workload correlations, structured event streams, or mode changes, a designer would need to provide the corresponding non-functional specifications. Having seen that calibration for real-time systems needs to be done with care and is time-consuming even when fully automated, it is questionable whether this is realistic in industrial scenarios. Second, in this thesis, basically complete knowledge of the hardware/software platform was assumed. Often, however, the knowledge about the implementation of hardware and software subsystems might be limited. In these cases, (overly) pessimistic assumptions about the behavior of these subsystems need to be made which deteriorates the tightness of the bounds obtained.
In conclusion, this thesis provides new insights into the potential and limitations of several techniques applicable in the context of design flows based on dataflow process networks. As for any new technique, the basis for drawing further conclusions will be the experience gained by using these techniques in a wide spectrum of applications. The encouraging results presented in this thesis could be motivation to do so.

6.3 Future Directions

Based on the conclusions drawn above, into which directions could future research go? Perhaps the most obvious is that it would be desirable to apply performance analysis to systems based on the proposed run-time environment, thereby creating a framework for designing efficient and at the same time best-case/worst-case analyzable systems. Basically, this could be done following the same approach as for the class of systems described in Chapter 5. Still, this requires the integration (and potentially also the development) of methods into Mπ that deal with back-pressure due to buffers with finite size in connection with cooperative scheduling. Furthermore, it would be interesting to investigate whether an architecture like the Cell Broadband Engine where a detailed hardware specification is not available could be accurately analyzed.

Even when progress in this direction is made, however, it is questionable whether efficiency and tight best-case/worst-case analysis can be achieved at the same time. Hence, one should perhaps distinguish between non real-time and real-time systems to identify further directions: the major goal for non real-time applications is to run efficiently on a wide range of multiprocessor systems. In analogy to the core premise of the Java programming language — “Write once, run anywhere” — one could phrase an analog vision — “Write once, run on any multiprocessor”. On the other hand, the major goal for real-time applications is to comply with the given timing constraints without relying on excessive over-dimensioning. Having seen the limitations of an approach relying on performance analysis (a posteriori), one could argue that it is better to guarantee real-time properties by construction (a priori). In the following, those two aspects are considered in some more detail.

Write once, run on any multiprocessor. For uniprocessors, one can write code once in a high-level language, such as C/C++ or Java, and execute it on every processor for which a suitable compiler or
virtual machine exists. For multiprocessors, software design typically resembles “embedded” software design, requiring intricate knowledge about the hardware architecture and low-level software libraries. In this thesis, it has been shown that, at least for streaming applications, automated software synthesis can yield efficient implementations, given the mapping of the application onto the architecture. Clearly, this requires that the application developer is able to come up with this mapping. A more subtle issue is that also the granularity of actors and the size of tokens communicated between them needs to be well chosen due to their big influence on the performance on a specific target architecture. To realize the vision sketched above, automated design space exploration and granularity control are required. Even though some initial progress in these directions has been made in design flows for model-driven development, such as STREAMiT ITKA02 and SEQUOIA FHK’06, further efforts are necessary to increase their applicability to the design of real systems. On the other hand, there is a large spectrum of online approaches supported by corresponding hardware units and run-time environments that reaches from conventional multi-threaded programming to new multiprocessor programming standards and languages like MCAPI, CUDA, and OpenCL. However, to obtain satisfactory results, even these approaches still rely on manual fine-tuning of code. “Write once, run on any multiprocessor” will thus stay an interesting research topic for the foreseeable future.

**Real-time by Construction.** Incomplete knowledge and complex interference of hardware and software components render the performance analysis of multiprocessor real-time systems a challenging task. While it is desirable to improve existing methods and investigate new ones, any uncertainty about a system implementation and non-determinism will always set a limit to performance analysis. A complementary approach is to construct systems in a way that they become amenable to analysis. Seminal in this respect are the time-triggered architecture Kop98 HHK03, various forms of fixed-priority and dynamic-priority servers But97, traffic shapers WMT06, and the controller area network (CAN) and FlexRay bus protocols. It is unlikely, however, that these approaches can be applied to multiprocessors in their original form considering that they were originally conceived for other application areas. As an example, establishing a global time base for a time-triggered approach might turn out to be difficult on a multiprocessor due to wiring constraints. Similarly, adding hardware traffic shapers to interconnect interfaces might cause a significant area overhead.

The challenge will thus be to design hardware/software architectures that achieve interesting trade-offs between predictability and efficiency.
Questions that need to be addressed in this context are, for instance: How much hardware support is required for constructing a hard real-time multiprocessor system? To what degree can software running on top of commercial (non real-time) multiprocessors enforce predictability? What are the appropriate programming models and corresponding analysis methods? Having seen the limitations of a purely analysis-based approach, finding answers to these questions seems to be crucial for the design of future real-time systems.

Irrespective of these two directions, the research agenda will be influenced by the multiprocessors that actually make it to the market and the concrete technical issues related to their application in different domains. Aspects that could gain importance in the domain of real-time systems are, for instance, energy-awareness, thermal-awareness, resilience against (temporal) failures of single processors or interconnect links, as well as their impact on hardware, software, and design methods. In any case, the design and performance analysis of multiprocessor streaming applications will continue to be an active research area.
Snapshots of the Design Flow of a Multiprocessor Streaming Application

This appendix lists the detailed DoD specification of the simple multiprocessor streaming application shown in Fig. 39 and the results of various design steps.

A.1 System Specification

This section lists the XML specifications of the process network, the architecture, and the mapping of the system depicted in Fig. 39. In addition, the C source code of the worker actor is shown as well as two possibilities to visualize these specifications.
A.1.1 Application Specification

Listing 6: XML source code of the dataflow process network shown in Fig. 39

```xml
<?xml version="1.0" encoding="UTF-8"?>
<processnetwork
xmlns="http://www.tik.ee.ethz.ch/shapes/schema/PROCESSNETWORK"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.tik.ee.ethz.ch/shapes/schema/PROCESSNETWORK
processnetwork.xsd"
name="producer-consumer">
  <process name="producer">
    <port type="output" name="1"/>
    <configuration name="triggerPeriod" value="2000000"/>
    <configuration name="jitter" value="1000000"/>
  </process>
  <process name="consumer">
    <port type="input" name="1"/>
    <source type="c" location="consumer.c"/>
  </process>
  <process name="worker">
    <port type="output" name="1"/>
    <source type="c" location="worker.c"/>
  </process>
  <sw_channel type="fifo" size="128" name="fifoA">
    <port type="input" name="0"/>
    <port type="output" name="1"/>
  </sw_channel>
  <sw_channel type="fifo" size="128" name="fifoB">
    <port type="input" name="0"/>
    <port type="output" name="1"/>
  </sw_channel>
  <connection name="producer-fifoA">
    <origin name="producer"> <port name="1"/> </origin>
    <target name="fifoA" > <port name="0"/> </target>
  </connection>
  <connection name="fifoA-worker">
    <origin name="fifoA" > <port name="1"/> </origin>
    <target name="worker" > <port name="1"/> </target>
  </connection>
  <connection name="worker-fifoB">
    <origin name="worker" > <port name="2"/> </origin>
    <target name="fifoB" > <port name="0"/> </target>
  </connection>
  <connection name="fifoB-consumer">
    <origin name="fifoB" > <port name="1"/> </origin>
    <target name="consumer" > <port name="1"/> </target>
  </connection>
</processnetwork>
```
A.1.2 C Source Code

Listing 7: C source code of the worker actor in Fig. 39. For each invocation, “worker” reads a floating point value from its input using `READ()`, squares it, and sends the result to its output using `WRITE()`.

```c
#define LENGTH 20
#define PORT_IN 1
#define PORT_OUT 2

typedef struct _local_states {
    int index;
    float i;
} WorkerState;

void worker_init(DOLProcess *);
int worker_fire(DOLProcess *);

void worker_init(DOLProcess *p) {
    p->local->index = 0;
}

int worker_fire(DOLProcess *p) {
    DOL_read((void*)PORT_IN, &(p->local->i),
              sizeof(float), p);
    p->local->i *= p->local->i;
    DOL_write((void*)PORT_OUT, &(p->local->i),
              sizeof(float), p);
    p->local->index++;
    if (p->local->index >= LENGTH) {
        DOL_detach(p);
    }
    return 0;
}
```

A.1.3 Architecture Specification

Listing 8: XML source code of the MPARM architecture shown in Fig. 39.

```xml
<architecture
    xmlns="http://www.tik.ee.ethz.ch/shapes/schema/ARCHITECTURE"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    name="mparm">
    <variable name="N" value="8"/>
    <iterator variable="i" range="N">
        <processor name="processor" type="RISC">
            <append function="i"/>
            <configuration name="frequency" value="100000000"/>
        </processor>
        <hw_channel name="processorbus" type="BUS">
            <append function="i"/>
            <configuration name="frequency" value="100000000"/>
        </hw_channel>
        <memory name="spm" type="RAM">
            <append function="i"/>
        </memory>
    </iterator>
</architecture>
```
A.1.4 Mapping Specification

Listing 9: XML source code of the mapping shown in Fig. 39.

```xml
<architecture>
  <iterator variable="i" range="N">
    <writepath name="spmtospm">
      <append function="i"/>
      <processor name="processor"/>
      <txbuf name="spm"/>
      <hw_channel name="processorbus"/>
    </writepath>
  </iterator>
  <iterator variable="j" range="N">
    <readpath name="spmfspsm">
      <append function="j"/>
      <processor name="processor"/>
      <chbuf name="spm"/>
      <hw_channel name="bus"/>
      <rxbuf name="spm"/>
    </readpath>
  </iterator>
</architecture>
```
A.1.5 Visualization of Specifications

![Diagram of process network](image)

**Figure 40**: The process network of Fig. 39 visualized in the Moses graphical editor. Based on this representation, the process network XML file and code templates of the actors can be automatically generated.
A.2 Software Synthesis

The following two listings show the results for running automated software synthesis to create a functional simulation based on protothreads.

Listing 10: C code of the FIRE() routine of the “worker” actor, as specified in Listing 7 transformed into a protothread by automated software synthesis.

```c
int worker_fire(DOLProcess *p)
{
    PT_BEGIN((pt*)(p->wpotr));

    PT_WAIT_UNTIL((pt*)(p->wpotr), read((void*)static_cast<workerWrapper*>(
        (static_cast<worker_data*>(p->wpotr)->wrapper)->port1Fifo,
        &(p->local->i), sizeof(float), p));
    p->local->i += p->local->i;

    PT_WAIT_UNTIL((pt*)(p->wpotr), write((void*)static_cast<workerWrapper*>(
        (static_cast<worker_data*>(p->wpotr)->wrapper)->port2Fifo,
        &(p->local->i), sizeof(float), p));
    p->local->index++;

    if (p->local->index >= LENGTH) {
        DOL_detach(p);
    }

    PT_END((pt*)(p->wpotr));
}
```
Listing 11: C code of a functional simulation based on protothreads, generated by automated software synthesis.

```c
#include "Fifo.h"
#include "producerWrapper.h"
#include "consumerWrapper.h"
#include "workerWrapper.h"

int main(void)
{
    Fifo fifoA(128);
    Fifo fifoB(128);

    int producerIndices[] = { -1, -1, -1, -1 };
    producerWrapper * producer = new producerWrapper("producer", producerIndices);
    int consumerIndices[] = { -1, -1, -1, -1 };
    consumerWrapper * consumer = new consumerWrapper("consumer", consumerIndices);
    int workerIndices[] = { -1, -1, -1, -1 };
    workerWrapper * worker = new workerWrapper("worker", workerIndices);

    producer->port1Fifo = &fifoA;
    consumer->port1Fifo = &fifoB;
    worker->port1Fifo = &fifoA;
    worker->port2Fifo = &fifoB;

    producer->init();
    consumer->init();
    worker->init();

    bool allBlocked = false;
    while (!allBlocked)
    {
        allBlocked = true;
        if (!producer->isDetached())
        {
            if (producer->fire() == PT_ENDED)
            {
                allBlocked = false;
            }
        }
        if (!consumer->isDetached())
        {
            if (consumer->fire() == PT_ENDED)
            {
                allBlocked = false;
            }
        }
        if (!worker->isDetached())
        {
            if (worker->fire() == PT_ENDED)
            {
                allBlocked = false;
            }
        }
    }

    delete producer;
    delete consumer;
    delete worker;

    return 0;
}
```
A.3 Automated Model Generation and Calibration

This section shows three results of automated performance analysis model generation and calibration for the multiprocessor streaming application shown in Fig. 39. A visualization of the execution trace on Mparm is shown in Fig. 42. The intermediate model is listed in Listing 12 and the corresponding Matlab Mscript in Listing 13.

**Figure 42:** Visualization of simulation trace on MParm, automatically extracted during model calibration. The trace is stored in value-change dump file (as standardized in the IEEE standard 1364-1995) and visualized using GTKWAVE [http://gtkwave.sourceforge.net]. For actors, “high” and “low” indicate when the thread of an actor is running. The ticks denote the beginning and end of `FIRE()`, `READ()`, and `WRITE()`. For channels, the number of tokens in the channel is shown.

**Listing 12:** Intermediate model of the process network in Fig. 39

```xml
<?xml version="1.0" encoding="UTF-8"?>
<analysismodel
  xmlns="http://www.tik.ee.ethz.ch/shapes/schema/ANALYSISMODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.tik.ee.ethz.ch/shapes/schema/ANALYSISMODEL
  http://www.tik.ee.ethz.ch/shapes/schema/analysismodel.xsd">
  <pjd name="trigger" period="200.000" jitter="1000.000" min_interarr="0.000"/>
  <task name="producer" bct="56.0000" wcet="66.0000">
    workload_lower="rtcurve([0 0.0 0.1 56.0 0.2 113.0 0.3 169.0 0; 4 226.0 0.5 282.0 0; 6 343.0 0.7 401.0 0.8 459.0 0; 9 517.0 0; 12 690.0 0.13 747.0 0; 14 805.0 0; 15 862.0 0.16 862.0 0.57])"
    workload_upper="rtcurve([0 66.0 0.1 129.0 0.2 191.0 0.3 254.0 0; 4 317.0 0.5 379.0 0; 6 442.0 0; 7 504.0 0.8 567.0 0.9 629.0 0; 10 662.0 0.9 724.0 0.1])"
  </task>
  <sink name="consumer_sink"/>
</analysismodel>
```
A.3. Automated Model Generation and Calibration

10 691.0 0:11 752.0 0:12 813.0 0:13 870.0 0:14 928.0 0;
15 985.0 (62.0))"/>
<task name="worker" bcet="56.0000" wcet="62.0000"
workload_lower="rtcurve([0 0.0 0.1 56.0 0.2 112.0 0.3 169.0 0;
4 225.0 0.5 284.0 0.6 341.0 0.7 399.0 0.8 457.0 0.9 514.0 0;
10 572.0 0.11 629.0 0.12 687.0 0.13 745.0 0.14 802.0 0;
15 860.0 0.16 860.0 0.57])"
workload_upper="rtcurve([0 62.0 0.1 121.0 0.2 180.0 0.3 239.0 0;
4 298.0 0.5 357.0 0.6 416.0 0.7 474.0 0.8 532.0 0.9 590.0 0;
10 648.0 0.11 705.0 0.12 763.0 0.13 821.0 0.14 878.0 0;
15 936.0 59.0])"/>
<task name="consumer" bcet="52.0000" wcet="62.0000"
workload_lower="rtcurve([0 0.0 0.1 52.0 0.2 105.0 0.3 158.0 0;
4 211.0 0.5 266.0 0.6 320.0 0.7 374.0 0.8 428.0 0.9 482.0 0;
10 536.0 0.11 590.0 0.12 643.0 0.13 697.0 0.14 751.0 0;
15 805.0 0.16 805.0 53.0])"
workload_upper="rtcurve([0 62.0 0.1 118.0 0.2 176.0 0.3 231.0 0;
4 290.0 0.5 345.0 0.6 400.0 0.7 459.0 0.8 514.0 0.9 573.0 0;
10 628.0 0.11 682.0 0.12 736.0 0.13 790.0 0.14 844.0 0;
15 898.0 56.0])"/>
<connection from="trigger" to="producer" scaling="1.0000"/>
<connection from="producer" to="worker" scaling="1.0000"/>
<connection from="worker" to="consumer" scaling="1.0000"/>
<resource name="processor_0" bandwidth="1.0000">
<fp preemptive="true">
<priority value="1">
<binding task="producer"/>
</priority>
<priority value="2">
<binding task="consumer"/>
</priority>
</fp>
</resource>
<resource name="processor_1" bandwidth="1.0000">
<fp preemptive="true">
<priority value="1">
<binding task="worker"/>
</priority>
</fp>
</resource>
<observation type="backlog" name="producer_backlog">
<attribute key="task" value="producer"/>
</observation>
<observation type="delay" name="producer_delay">
<attribute key="task" value="producer"/>
</observation>
<observation type="backlog" name="worker_backlog">
<attribute key="task" value="worker"/>
</observation>
<observation type="delay" name="worker_delay">
<attribute key="task" value="worker"/>
</observation>
<observation type="backlog" name="consumer_backlog">
<attribute key="task" value="consumer"/>
</observation>
<observation type="delay" name="consumer_delay">
<attribute key="task" value="consumer"/>
</observation>
<observation type="latency" name="triggerconsumer_latency">
<attribute key="from" value="trigger"/>
<attribute key="to" value="consumer"/>
</observation>
</analysismodel>
Listing 13: Matlab MPA script for the process network in Fig. 39.

```matlab
trigger_out = rtcpjd(200.0000, 100.0000, 0.0000);
producer_demand = [66.0000 56.0000];
producer_workload = [...
4 317.0 0.5 379.0 0.6 442.0 0.7 504.0 0.8 567.0 0.9 629.0 0; ...
10 691.0 0.11 752.0 0.12 813.0 0.13 870.0 0.14 928.0 0; ...
15 985.0 62.0)];
rtcurve([0.00 0.1 56.0 0.2 113.0 0.3 169.0 0; ...
4 226.0 0.5 282.0 0; 6 343.0 0.7 401.0 0.8 459.0 0.9 517.0 0; ...
10 575.0 0.11 632.0 0; 12 690.0 0.13 747.0 0.14 805.0 0; ...
15 862.0 0.16 862.0 57.0)];
worker_demand = [62.0000 56.0000];
worker_workload = [...
4 298.0 0.5 357.0 0.6 416.0 0.7 474.0 0.8 532.0 0.9 590.0 0; ...
10 648.0 0.11 705.0 0.12 763.0 0.13 821.0 0.14 878.0 0; ...
15 936.0 59.0); ...
rtcurve([0.0 0.1 56.0 0.2 112.0 0.3 169.0 0; ...
4 225.0 0.5 284.0 0; 6 341.0 0.7 399.0 0.8 457.0 0.9 514.0 0; ...
10 572.0 0.11 629.0 0; 12 687.0 0.13 745.0 0.14 802.0 0; ...
15 860.0 0.16 860.0 57.0)]);
consumer_demand = [62.0000 52.0000];
consumer_workload = [...
4 290.0 0.5 345.0 0.6 400.0 0.7 459.0 0.8 514.0 0.9 573.0 0; ...
10 628.0 0.11 682.0 0.12 736.0 0.13 790.0 0.14 844.0 0; ...
15 898.0 56.0]); ...
rtcurve([0.0 0.1 52.0 0.2 105.0 0.3 158.0 0; ...
4 211.0 0.5 266.0 0; 6 320.0 0.7 374.0 0.8 428.0 0.9 482.0 0; ...
10 536.0 0.11 590.0 0.12 643.0 0.13 697.0 0.14 751.0 0; ...
15 805.0 0.16 805.0 53.0)]);
b1 = rtcfs(1.0000);
b4 = rtcfs(1.0000);
[producer_out b2 producer_delay producer_buf] = ...
rtgpwl(trigger_out, b1, producer_workload);
[worker_out b5 worker_delay worker_buf] = ...
rtgpwl(producer_out, b4, worker_workload);
consumer_sink = consumer_out;
producer_backlog = producer_buf;
producer_delay = producer_delay;
worker_backlog = worker_buf;
worker_delay = worker_delay;
consumer_backlog = consumer_buf;
consumer_delay = consumer_delay;
triggerconsumer_latency = producer_delay + worker_delay + consumer_delay;
```
Bibliography


137

Bibliography


[SBW09] Marcel Steine, Marco Bekooij, and Maarten Wiggers. A Priority-Based Scheduler with Conservative Dataflow


Curriculum Vitae

Wolfgang Haid was born on November 24, 1980 in Linz, Austria. He received his BSc degree in telematics from the Technical University of Graz, Austria, and his MSc degree in electrical engineering from ETH Zurich, Switzerland, in 2004 and 2006, respectively. Afterwards, he joined the Computer Engineering and Networks Laboratory at ETH Zurich as a research and teaching assistant, working on techniques for the design of multiprocessor streaming applications. During his time at ETH Zurich, he received a best paper award at the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS) and a best paper award nomination at the IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), both in 2009. His research interests include software development and performance analysis for multiprocessors, especially for real-time streaming applications.